Die Java Pattern Language

Eine graphbasierte Sprache zur Spezifikation komplexer Suchmuster für die statische Quelltextanalyse

Dissertation

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften
(Dr. rer. nat)

durch die
Fakultät für Wirtschaftswissenschaften
Institut für Informatik und Wirtschaftsinformatik
Universität Duisburg-Essen
Campus Essen

vorgelegt von
Dipl.-Wirt.Inf. Carsten Köllmann
geboren in Essen

Essen, 2014

Tag der mündlichen Prüfung: 12.02.2015

Erstgutachter: Prof. Dr. Michael Goedicke

Zweitgutachterin: Prof. Dr. Gabriele Taentzer

Inhaltsverzeichnis

Inhaltsverzeichnis ..... **iii**
Abbildungsverzeichnis ..... **v**
Tabellenverzeichnis ..... **viii**
Abkürzungsverzeichnis ..... **ix**

1. Einleitung **1**
1.1. Statische Analyse zur Mustersuche im Quellcode 1
1.2. Anforderungen an eine Spezifikationssprache zur Mustererkennung in Quelltexten 3
1.3. Beitrag der Arbeit 6
1.4. Aufbau der Arbeit 9
2. Verwandte Arbeiten **11**
2.1. Techniken zur automatisierten Mustererkennung im Quellcode 11
2.2. Analyse von Programmierübungen 13
2.3. Suche nach Pattern im Quellcode zum Programmverstehen 16
2.4. Suche nach Programmstrukturen zum Refaktoring 18
2.5. Hierarchische und modulbasierte Ansätze zur Graphtransformation 20
2.6. Erweiterung der bestehenden Arbeiten durch die JPL 22
2.7. Zusammenfassung 24
3. Quelltextrepräsentation **25**
3.1. Abstrakter Syntaxbaum 26
3.2. Java Code Graph 27
3.3. Adapted Java System Dependency Graph 31
3.4. Transitive Hülle 37
3.5. Pfad 38
3.6. Zusammenfassung 40
4. Java Pattern Language **41**
4.1. Überblick über die Syntax der JPL 42
4.2. Überlegungen zur Modularität 45
4.3. Modulrepräsentation der JPL 47
4.4. Die JPL-Schnittstellenstruktur 50
4.5. Modulkopplung über Datenabhängigkeiten 52
4.6. Repräsentation hierarchischer Strukturen 56
4.7. Kopplung semantisch zusammenhängender Module 57
4.8. Beispiel zur Kopplung über JCG-Elemente 58
4.9. Beispiel zur Kopplung über AJSDG-Elemente 61
4.10. Zusammenfassung 63
5. Ableitung des JPL-Graphen und Durchführung der Mustersuche **65**
5.1. Graphtransformation im Software Engineering 67
5.2. Grundlagen der Graphtransformation 68
5.3. Vorgehensweisen zur Ableitung eines JPL-Musters 73
5.4. Ableitung der JPL-Schnittstellenstrukturen 77
5.5. Varianten für die Variablenübergabe durch direkte Aufrufe 82
5.6. Varianten für die Variablenübergabe durch indirekte Aufrufe 92
5.7. Erstellung der Datei zur Mustersuche bei separater Sektionsbetrachtung 99
5.8. Transitive Hülle 103

5.9. Ausführung der Mustererkennung für ein einzelnes JPL-Modul 105
5.10. Sektionsbasierte Mustersuche für JPL-Muster 117
5.11. Slicing 122
5.12. Zusammenfassung 124
6. Vorgehensweisen für die Spezifikation von JPL- Mustern 125
6.1. Top-Down ausgehend von funktionalen Anforderungen 126
6.2. Bottom-Up über Syntaxanalyse 129
6.3. Bottom-Up über Variablenabhängigkeitsbetrachtung 130
6.4. Bottom-Up mittels eines gegebenen Musterkataloges 132
6.5. Zusammenfassung 132
7. Automatisierte Testumgebung 134
7.1. Arbeitsablauf für die Prüfung von Übungslösungen 134
7.2. Systemumgebung zur Unterstützung der Spezifikation von Suchmustern 135
7.3. Automatisierter Ablauf der Mustersuche 156
7.4. Zusammenfassung 158
8. Anwendungsbeispiele 159
8.1. Mathematische Funktion 160
8.2. Lager als Liste 165
8.3. Geometrische Objekte 168
8.4. Verwaltung geometrischer Objekte als Liste 171
8.5. Kriterien zu geeigneten Übungsaufgaben und Hinweise zur Erstellung von Suchmustern 173
8.6. Zusammenfassung 176
9. Fazit 177
9.1. Beitrag der Arbeit 177
9.2. Kritische Analyse 179
9.3. Ausblick 180
10. Literaturverzeichnis 182
11. Anhang 191
11.1. Syntax des Java Code Graphen 192
11.2. Unterstützende Graphregeln 206
11.3. Regelzuordnung von Anwendungsbeispielen zur Dissertation 223
11.4. Inhalt der beiliegenden DVD 240

Abbildungsverzeichnis

Abbildung 1: Graphtypen zur Quelltextrepräsentation 25
Abbildung 2: Zuordnung eines Parameters zu einem Methodenkopf im JCG 28
Abbildung 3: JCG-Graph zu den Klassen *Student* und *Studentlist* 30
Abbildung 4: Intraprozeduraler Abstrakter Java Systemabhängigkeitsgraph 35
Abbildung 5: Java Systemabhängigkeitsgraph mit objektorientierten Strukturen 36
Abbildung 6: Transitive Hülle über dem Datenabhängigkeitsgraphen 37
Abbildung 7: Transitive Hülle über dem Kontrollabhängigkeitsgraphen 38
Abbildung 8: Suchmuster zur Erläuterung des Pfad-Elements in der JPL 39
Abbildung 9: Syntaxelemente und deren Verwendungskontexte in der JPL 41
Abbildung 10: Beziehung zwischen JCG und AJSDG 45
Abbildung 11: Komponentendarstellung in II 46
Abbildung 12: Moduldarstellung der JPL 47
Abbildung 13: Kopplung über JCG. Direkter Zugriff auf eine Variable 53
Abbildung 14: Kopplung über JCG: Indirekter Zugriff über Methodenaufruf 53
Abbildung 15: Kopplung über JCG: Indirekter Zugriff über return-Ausdruck 54
Abbildung 16: Beispiel für die Verwendung von JPL-Schnittstellenknoten 55
Abbildung 17: Kopplung über AJSDG-Knoten 55
Abbildung 18: Hierarchiebeziehungen in der JPL 56
Abbildung 19: Repräsentation einer *if...then* Bedingung 57
Abbildung 20: Suchmuster für Variante 1: Belegen der Variablen 58
Abbildung 21: Suchmuster zu Variante 1: Ausgabe der Variablen 59
Abbildung 22: Suchmuster zu Variante 2: Belegen und Ausgabe der Variablen 59
Abbildung 23: Suchmuster unter Verwendung von JPL-Schnittstellenknoten 60
Abbildung 24: JPL-Spezifikation zu Anforderung 1 61
Abbildung 25: JPL-Spezifikation zu Anforderung 2 62
Abbildung 26: JPL-Spezifikation zu Anforderung 3 62
Abbildung 27: Prozess der Ableitung des JPL-Graphen in Graphtransformationsregeln sowie Ausführung der Mustersuche 66
Abbildung 28: Schematische Darstellung der direkten Ableitung als Pushout 70
Abbildung 29: Darstellung der direkten Ableitung im DPA als „Klebegraph“ 71
Abbildung 30: Ableitung der modulbasierten JPL-Spezifikation zum JPL-Graphen 73
Abbildung 31: Allgemeine Regel zur Ableitung der JPL-Schnittstellenknoten 78
Abbildung 32: Regelmenge zur Auflösung der JPL-Schnittstellenknoten 82
Abbildung 33: Regelvarianten zur Analyse des exportierenden Gültigkeitsbereichs einer Membervariablen 84
Abbildung 34: NAC zur Gültigkeitsbereichsüberprüfung 85
Abbildung 35: Regelvarianten zur Ermittlung des Gültigkeitsbereichs eines Variablenaufrufs 86
Abbildung 36: Regelvarianten zur Ermittlung des Gültigkeitsbereichs eines Variablenaufrufs bei verbundenem Referenzknoten 88
Abbildung 37: Regel zur Suche exportierender lokaler Gültigkeitsbereiche 90
Abbildung 38: Regel zur Suche ex- und importierender lokaler Gültigkeitsbereiche 90
Abbildung 39: Änderung des Typs JPLPrimitive in JPLPrimitiveDeclaration 92
Abbildung 40: Regel zur Erstellung des Suchmusters der Parameterübergabe 93
Abbildung 41: Regel zur Erstellung des Suchmusters der Übergabe durch Objektinstanziierung 94
Abbildung 42: Regel zur Erstellung des Suchmusters der Übergabe durch *return* 95

Abbildung 43: Suchmuster der Übergabe durch *Getter/Setter-Methoden* 96
Abbildung 44: Regel zur Erstellung des Suchmusters für eine Setter-Methode 96
Abbildung 45: Regel zur Erstellung des Suchmusters für eine Getter-Methode 97
Abbildung 46: Regel zur Erstellung des Suchmusters der Übergabe mittels einer hierarchisch übergeordneten Variablen 98
Abbildung 47: Regel zur Erstellung des Teilmusters, welches den Graphen der Körpersektion umfasst 100
Abbildung 48: Regel zur Erstellung des Teilmusters, welches den Graphen zur Übergabestruktur umfasst 100
Abbildung 49: Regel 1 zur Erstellung der transitiven Hülle für Datenabhängigkeiten 104
Abbildung 50: Regel 2 zur Erstellung der transitiven Hülle für Datenabhängigkeiten 104
Abbildung 51: Muster zur Identifikation des Gesamtmusters 108
Abbildung 52: Graph eines Einzelmusters nach Schritt 5 des Algorithmus 109
Abbildung 53: Graph eines Einzelmusters nach Ablauf des Algorithmus 110
Abbildung 54: Durchführung der Suche nach dem Pfad Element 1 113
Abbildung 55: Durchführung der Suche nach dem Pfad Element 2 113
Abbildung 56: Durchführung der Suche nach dem Pfad Element 3 114
Abbildung 57: Durchführung der Suche nach dem Pfad Element 4 114
Abbildung 58: Durchführung der Suche nach dem Pfad Element 5 115
Abbildung 59: Beispiel für die Suche nach einem Pfad 116
Abbildung 60: Beispiel zur Suche nach komplexen JPL-Mustern 1 118
Abbildung 61: Beispiel zur Suche nach komplexen JPL-Mustern 2 119
Abbildung 62: Muster zur Identifikation des Gesamtmusters 120
Abbildung 63: Beispiel zur Suche nach komplexen JPL-Mustern 3 121
Abbildung 64: Regel 1 zur Generierung eines Forward-Slices 123
Abbildung 65: Regel 2 zur Generierung eines Forward-Slices 123
Abbildung 66: Regel 3 zur Generierung eines Forward-Slices 123
Abbildung 67: Spezifikation von Datenabhängigkeiten 130
Abbildung 68: Spezifikation mit zusätzlicher JCG-Struktur 131
Abbildung 69: Arbeitsablauf zur automatisierten Prüfung von Übungsaufgaben 135
Abbildung 70: Prozess zur Erstellung der Datei, über welche die Mustersuche durchgeführt wird 139
Abbildung 71: JPL-Graph des Suchmusters 147
Abbildung 72: Aus dem JPL-Graphen abgeleitete Suchmustervariante 148
Abbildung 73: Regelsätze zur Gesamtmustersuche, gezeigt: Regel zur Bereichsmarkierung 149
Abbildung 74: Regel zur Suche des Gesamtmusters 149
Abbildung 75: JPL-Graph des Suchmusters 150
Abbildung 76: Aus dem JPL-Graphen abgeleitete Suchmustervariante 151
Abbildung 77: Regelsätze zur sektionsbasierten Mustersuche, gezeigt: Regel zur Bereichsmarkierung 153
Abbildung 78: Regel zur Suche nach einer Körperstruktur 153
Abbildung 79: Regel zur Suche nach einer Schnittstellenstruktur 154
Abbildung 80: Aus dem JPL-Graphen abgeleiteter Gesamtgraph (rechts) 154
Abbildung 81: Workflow auf dem Server zur automatisierten Prüfung von Übungsaufgaben. 156
Abbildung 82: Mathematische Funktion, JPL-Muster 1. 161
Abbildung 83: Mathematische Funktion, JPL-Muster 2. 163
Abbildung 84: Lebensmittellager, JPL-Muster 3 166

Abbildung 85: Geometrisches Objekt, JPL-Muster 4. 169
Abbildung 86: Geometrische Objekte über Liste, JPL-Muster 5. 172
Abbildung 87: Bereichsmarkierung für Bereiche innerhalb von Klassen 1 206
Abbildung 88: Bereichsmarkierung für Bereiche innerhalb von Klassen 2 206
Abbildung 89: Bereichsmarkierung für Klassen 1 207
Abbildung 90: Bereichsmarkierung für Klassen 2 207
Abbildung 91: Bereichsmarkierung mit hierarchischen Abhängigkeiten 208
Abbildung 92: Bereichsmarkierung für ASDG-Knoten 208
Abbildung 93: Markierung nicht zum Bereich gehörender Knoten mit Bereichsmarkierungsknoten 209
Abbildung 94: Markierung nicht zum Bereich gehörender Knoten über Attribut *ModulNr* 209
Abbildung 95: Markierung möglicher Quellknoten für die Pfadsuche 209
Abbildung 96: Markierung möglicher Zielknoten für die Pfadsuche 210
Abbildung 97: Initiale Erstellung eines Bereichsmarkierungsknotens mit Verbindung zum *MarkAll*-Knoten 210
Abbildung 98: Markierung der JCG/AJSDG-Knoten in einem Modul 210
Abbildung 99: Markierung der JPL-Schnittstellenknoten: direkte Übergabe 1 211
Abbildung 100: Markierung der JPL-Schnittstellenknoten: direkte Übergabe 2 211
Abbildung 101: Markierung der JPL-Schnittstellenknoten: indirekte Übergabe 211
Abbildung 102: Markierung der Knoten außerhalb des Gültigkeitsbereichs 1 211
Abbildung 103: Markierung der Knoten außerhalb des Gültigkeitsbereichs 2 212
Abbildung 104: NAC-Variante zum *MarkAll*-Knoten 212
Abbildung 105: Entfernen der Kanten zum *accessToPrimitiveTypeVariable*-Knoten 212
Abbildung 106: Entfernen doppelter Kanten zum *Bereichsmarkierungsknoten* 213
Abbildung 107: Entfernen des *accessToPrimitiveTypeVariable*-Knotens 213
Abbildung 108: Umhängen auf den *JPLPrimitiveDeclaration*-Knoten 214
Abbildung 109: Umbenennung der *JPLPrimitiveDeclaration* 214
Abbildung 110: Ableitung von direkten JCG-Übergabestrukturen 1 214
Abbildung 111: Ableitung von direkten JCG-Übergabestrukturen 2 214
Abbildung 112: Ableitung von direkten AJSDG-Übergabestrukturen 1 215
Abbildung 113: Ableitung von direkten Übergabestrukturen 2 215
Abbildung 114: Ableitungsvariante für Muster mit verbundenen Referenzknoten 1 216
Abbildung 115: Ableitungsvariante für Muster mit verbundenen Referenzknoten 2 216
Abbildung 116: Ableitungsvariante für Muster mit verbundenen Referenzknoten 3 217
Abbildung 117: Ableitungsvariante für Muster mit verbundenen Referenzknoten 4 217
Abbildung 118: Ableitungsvariante für Muster mit verbundenen Referenzknoten 5 218
Abbildung 119: Einfügen der Schnittstellenstruktur in das Gesamtmuster 218
Abbildung 120: Einfügen der Struktur zur Markierung der Körpersektion 1 219
Abbildung 121: Einfügen der Struktur zur Markierung der Körpersektion 2 219
Abbildung 122: Initialer Ableitungsschritt bei JPL-Schnittstellenstruktur 220
Abbildung 123: Regel zur Erstellung des JCG: Kante zur Variablenrückgabe 220
Abbildung 124: Regel zur Erstellung des JCG: Vererbungskante 221
Abbildung 125: Hilfsknoten an JCG-Knoten 221
Abbildung 126: Normalisierung der Hilfsknoten 1 221
Abbildung 127: Normalisierung der Hilfsknoten 2 222

Tabellenverzeichnis

Tabelle 1: Regeln zur Ableitung des JPL-Graphen für „Aufgabe 1 Muster 1 Parameter eine Bedingung“ 224
Tabelle 2: Regeln zur Ausführung der Mustersuche für „Aufgabe 1 Muster 1 Parameter eine Bedingung“ 225
Tabelle 3: Regeln zur Ableitung des JPL-Graphen für „Aufgabe 1 Muster 2 GlobalVar“ 226
Tabelle 4: Regeln zur Ausführung der Mustersuche für „Aufgabe 1 Muster 2 GlobalVar“ 226
Tabelle 5: Regeln zur Ausführung der Mustersuche für „Aufgabe 1 Muster 2 GlobalVar mit NAC“ 227
Tabelle 6: Regeln zur Ableitung des JPL-Graphen für „Aufgabe 1 Muster 2 Parameterübergabe zwei Bedingungen“ 228
Tabelle 7: Regeln zur Ausführung der Mustersuche für „Aufgabe 1 Muster 2 Parameterübergabe zwei Bedingungen“ 229
Tabelle 8: Regeln zur Ableitung des JPL-Graphen für „Aufgabe 2 Muster 3 Lebensmittel GlobalVar“ 230
Tabelle 9: Regeln zur Ausführung der Mustersuche für „Aufgabe 2 Muster 3 Lebensmittel GlobalVar“ 231
Tabelle 10: Regeln zur Ableitung des JPL-Graphen für „Aufgabe 3 Muster 4 Vererbung“ 232
Tabelle 11: Regeln zur Ausführung der Mustersuche für „Aufgabe 3 Muster 4 Vererbung“ 232
Tabelle 12: Regeln zur Ableitung des JPL-Graphen für „Aufgabe 4 Muster 5 Lokale Variable“ 234
Tabelle 13: Regeln zur Markierung von Teilgraphen des JPL-Graphen für „Aufgabe 4 Muster 5 Lokale Variable“ 234
Tabelle 14: Regeln zur Vorbereitung des Kopiervorgangs von Teilgraphen des JPL-Graphen für „Aufgabe 4 Muster 5 Lokale Variable“ in die Datei zum Ablauf der Mustersuche. 235
Tabelle 15: Template zur Erstellung des Gesamtmusters für die sektionsbasierte Suche 235
Tabelle 16: Regeln zur Ausführung der Mustersuche für „Aufgabe 4 Muster 5 Lokale Variable“ 237

Abkürzungsverzeichnis

AGG	Attributed Graph Grammar
AJSDG	Adapted Java System Dependency Graph
AJSDG_SK	AJSDG Schnittstellenknoten
AST	Abstract Syntax Tree
ASG	Abstrakter Syntax Graph
CDG	Class Dependence Graph
DPA	Double Pushout Ansatz
JCG	Java Code Graph
JCG_SK	JCG Schnittstellenknoten
JPL	Java Pattern Language
JPLG	JPL Graph
JPLM	JPL Module
JPLSK	JPL Schnittstellenknoten
MVBKa	Modulverbindungskanten
MVBKaS	Modulverbindungskanten für semantisch zusammengehörende Blöcke
NAC	Negative Application Condition
PDG	Program Dependency Graph
SDG	System Dependency Graph
SKa	Schnittstellenkante
SPA	Single Pushout Ansatz
UML	Unified Modeling Language
XML	Extensible Markup Language
XPath	XML Path Language
XSLT	Extensible Stylesheet Language Transformation

1. Einleitung

Das Ziel dieser Arbeit ist die Darstellung einer graphbasierten Sprache für die Beschreibung komplexer Suchmuster zur Quelltextanalyse. Der Fokus liegt auf der Möglichkeit einer modularisierten Musterspezifikation, so dass der Nutzer komplexe Muster über die Kombination einzelner Module erstellen kann. Die Sprachsyntax basiert sowohl auf dem *Abstrakten Syntaxbaum*, als auch auf Elementen der Graphtypen *Kontrollabhängigkeitsgraph* und *Datenabhängigkeitsgraph*. Somit kann in einer Musterspezifikation die Graphrepräsentation von Java Syntaxelementen mit abstrakteren Abhängigkeitsbeziehungen kombiniert werden.

Die Suche nach komplexen Mustern zur Quelltextanalyse wird in verschiedenen Bereichen angewendet, u.a. im Rahmen von Sicherheitsaudits (Suche nach bösartigem Quellcode, Viren, Trap Doors, etc.), dem Quellcode-Refaktoring, der Bewertung von Quelltext bezüglich dessen Komplexität, zur Unterstützung des Verstehens von unzureichend dokumentiertem Quelltext, als auch allgemein dessen Wartung. Die Domäne, für die der hier vorgestellte Ansatz entwickelt wurde, liegt in der Analyse studentischer Übungsaufgaben zur Bewertungs- und Korrekturunterstützung. Die Technik lässt sich allerdings auch auf die zuvor genannten Bereiche übertragen.

Dieses Kapitel beschreibt in Abschnitt 1.1. den Kontext der Arbeit und vertieft in Abschnitt 1.2. die domänenspezifischen Herausforderungen, welche im Fokus des Forschungsbeitrags liegen. In Abschnitt 1.3. werden die Beiträge der Arbeit im Überblick dargestellt, während in Abschnitt 1.4. der Aufbau der Arbeit erläutert wird.

1.1. Statische Analyse zur Mustersuche im Quellcode

Die statische Quelltextanalyse gehört im Bereich des Testens und der Wartung von Softwaresystemen zu den am häufigsten verwendeten Techniken, um fehlerhafte Strukturen zu identifizieren und das Codeverstehen zu unterstützen. Neben den manuellen Techniken der Inspektion und des Code-Walkthrough, haben sich verstärkt toolbasierte Analyseverfahren etabliert, welche auf der Suche nach Mustern auf Architektur- oder Quelltextebene basieren. Diese Applikationen leiten die zu suchenden Muster z.B. aus Katalogen von typischen Programmierfehlern und Style Guides für Programmierer ab. Hinzu kommen Templates für spezifische Analysen, wie die Suche nach typischen sicherheitskritischen Stellen im Quellcode (Trap Doors, Trojaner, etc.), oder nach performanceintensiven Codefragmenten, welche z.B. eine Vielzahl von Kommunikationsverbindungen etablieren oder komplexe Berechnungen initiieren.

Während diese Suchmuster häufig unabhängig vom jeweiligen Kontext der zu untersuchenden Applikation definiert werden, sind im Weiteren Techniken entwickelt worden, welche, die Existenz von kontextabhängigen Strukturen nachweisen. Hierbei werden keine fehlerhaften, sondern für die Funktionalität des Systems notwendige Muster gesucht. Somit können durch diese Technik verstärkt semantische Aspekte von Quellcode im Kontext gegebener Anforderungen analysiert werden. Angewendet werden diese Techniken u.a. in der Analyse von Übungsaufgaben [KG06][LCH+06], der Identifikation von Services [Mey00] oder auch bei der Suche nach Design-Pattern zur Erleichterung des Codeverständnisses [BBS05].

Mustererkennungsansätze über Codestrukturen können auf verschiedenen Abstraktionsebenen eingesetzt werden. So fokussieren sich einige Arbeiten auf Softwaresysteme im Großen [Wen05], [NWW03], ohne Einbezug der Detailebene. Andere Ansätze unterstützen die Suche nach Softwarepattern mit Fokus auf Klassen- und Methodenebene [DMT10] oder sie betrachten einfache Codestrukturen, welche nur wenig semantische Informationen enthalten [BPT04]. Das Ziel dieser Arbeit ist die Erstellung einer graphbasierten visuellen Sprache, mit Hilfe derer Strukturen definiert und analysiert werden können, deren Spezifikation sowohl die Detailebene als auch höhere Abstraktionsebenen mit einbezieht. Die Spezifikation der Suchmuster erfolgt über Graphen, da Beziehungen zwischen einzelnen Musterelementen hier direkt abgebildet werden können und Reihenfolgen einzelner zu suchender Quellcodestrukturen nicht angegeben werden müssen.

Die Vorgehensweise der Suchmustererstellung basiert auf der strukturierten Komposition einfacher Suchmuster zu komplexen Mustern. Die Suchmuster werden in einzelne Module unterteilt, um dem Problem der Unübersichtlichkeit großer Graphen zu begegnen. Das Ergebnis dieser Arbeit ist die Suchmuster-Spezifikationssprache: *Java Pattern Language (JPL)*. Die Sprache Java wurde auf Grund ihrer weiten Verbreitung für diese Arbeit ausgewählt. Die grundlegende Vorgehensweise und die Technik dieses Ansatzes sind auch auf andere Programmiersprachen übertragbar.

Als Domäne zur Anwendung der JPL wird die Analyse von Programmierübungen betrachtet. Die JPL wird hier eingesetzt, um Lösungsmuster zu den gegebenen strukturellen Anforderungen der Übungsaufgaben zu spezifizieren. Die Ableitung der Muster (ein JPL-Muster kann mehrere konkrete Ausprägungen von Suchmuster beinhalten) basiert auf Graphtransformationsregeln, welche die abstrakten JPL-Suchmuster in konkrete Suchmuster über Quelltextelemente transformieren. Diese Strukturen werden unter Verwendung einer Toolumgebung automatisch in der Graphrepräsentation der Quelltexte der Übungslösungen gesucht, so dass sich der Korrekturaufwand im Gegensatz zur manuellen Analyse verringert. Auch für die Mustersuche wird Graphtransformation eingesetzt, da durch Transformationsregeln Hilfsstrukturen in der Quelltextrepräsentation erstellt werden, die für die Mustersuche notwendig sind. Diese umfassen die Markierung der Quellcodebereiche in denen nach dem Muster gesucht werden soll und die Markierung von Teilmustern. Weiterhin werden für die Suche nach Daten- und Kontrollabhängigkeiten Strukturen erstellt, so diese im Suchmuster spezifiziert wurden.

Weitere Domänen mit ähnlichen Charakteristika, in denen die JPL eingesetzt werden kann, sind z.B. die Suche nach sicherheitskritischen Codestrukturen, die Ermittlung von Ansatzpunkten für Refaktorings oder die Suche nach Design Pattern zur Unterstützung des Quelltextverständnisses.

Im folgenden Kapitel werden die Probleme und Anforderungen der Domäne: „Mustererkennung in Quelltext“ näher beschrieben, um nachfolgend den Beitrag dieser Arbeit herauszuarbeiten.

1.2. Anforderungen an eine Spezifikationssprache zur Mustererkennung in Quelltexten

In diesem Kapitel wird die Problemdomäne der Suchmusterspezifikation für Quelltextanalysen weitergehend eingegrenzt, um nachfolgend die Anforderungen herauszuarbeiten, welchen die in dieser Arbeit dargestellte Spezifikationssprache JPL genügen muss.

Beschreibung der Problemdomäne:
Die betrachtete Domäne umfasst den Prozess der statischen Mustererkennung von der Suchmusterspezifikation über die Durchführung der Suche bis zur Ergebnisanalyse. Im Speziellen wird die Anwendung von Suchmustern über den Quellcode von Lösungen für Übungsaufgaben, welche im Rahmen von Programmierungsübungen gestellt werden, untersucht. Ziel der Mustererkennung ist es, die vom Übungsleiter spezifizierten und aus den Anforderungen abgeleiteten Muster im Quelltext zu identifizieren, um somit die Übungskorrektur zu unterstützen. Die Prüfung des Quelltextes basiert auf der Suche nach Strukturen, welche für die Lösung notwendig sind, d.h. es werden kontextsensitive Suchmuster spezifiziert. Werden diese Muster nicht gefunden, so ist es wahrscheinlich, dass die Lösung fehlerhaft ist und ein entsprechender Hinweis wird generiert.

Domänenspezifische Herausforderungen:

- Der Aufwand der Musterspezifikation zu Anforderungen, welche durch eine große Anzahl an Varianten implementiert werden können, ist sehr hoch.
- Durch Abstraktion vom Quelltext verringert sich der zuvor beschriebene manuelle Arbeitsaufwand. Allerdings erhöht sich die Gefahr, dass inkorrekte Lösungen fälschlich als korrekt bewertet werden (*false positives*).
- Für die Abfrage von Anforderungen, deren Implementierung sich über mehrere Klassen oder Methoden erstreckt, müssen komplexe Muster spezifiziert werden, die schnell unübersichtlich werden können, so dass die Wahrscheinlichkeit von Fehlern in der Spezifikation mit zunehmendem Musterumfang steigt.
- Die Spezifikation von Mustern erfolgt in der Praxis vielfach auf Textebene, wie z.B. über reguläre Ausdrücke. Die Anwendung dieser Technik ist wenig intuitiv und hierdurch fehleranfällig.
- Die grundlegende Konsistenz eines Suchmusters muss gewährleistet sein. So müssen z.B. bei einer graphbasierten Spezifikation eines Suchmusters die Kanten, welche die Beziehung zwischen zwei Quelltextelementen im Suchmustergraphen beschreiben, auch immer mit zwei Knoten, welche diese Elemente repräsentieren, verbunden sein.

Hypothesen auf denen die vorliegende Arbeit beruht:

- Beziehungen zwischen einzelnen Codestrukturen sind wichtig für die Formulierung von Suchmustern und können visuell in einem Graphen über Kanten direkter beschrieben werden als über Abfragen auf Textebene.
- Durch die gleichzeitige überlagernde Verwendung von Elementen unterschiedlicher Abstraktionsebenen in Suchmustern können diese genauer beschrieben werden, als dies bei Verwendung nur einer Beschreibungsebene möglich ist. So können z.B. durch die Kombination von JCG (Java Code Graph, eine Erweiterung des Abstrakten Syntaxbaums, zu Details s. Kapitel 3.2) und einer Datenflussstruktur diese verschiedenen Sichten auf den Quelltext gemeinsam abgebildet, und das Muster somit kontextgenauer spezifiziert werden, als wenn nur eine Ebene verwendet würde. Über die Elemente des JCG werden syntaktische Elemente des Quelltextes dargestellt, welche in der Lösung genutzt werden müssen. Die Datenflussstruktur abstrahiert von Elementen des Quelltextes und beschreibt Beziehungen zwischen Ausdrücken. Wenn sich Variablen des über den JCG spezifizierten Suchmusters mit Variablen des über den Datenfluss dargestellten Musters überlagern, entsteht ein kombiniertes Muster, welches zwei Abstraktionsebenen erfasst. Der Vorteil liegt darin, dass erstens mögliche *false positives*, die über ein reines Datenabhängigkeitssuchmuster identifiziert würden, durch eine zusätzliche Musterkonkretisierung mit JCG-Elementen vermieden werden, und dass zweitens über den JCG nur aufwändig zu beschreibende Beziehungen zwischen Elementen des Quelltextes über den Datenfluss einfacher spezifiziert werden können.
- *False positives* können besser erkannt werden, wenn anstatt einer einfachen Erfolgsmeldung nach der Mustersuche der Kontext, in dem das Muster gefunden wurde, näher erläutert wird.
- Die Spezifikation komplexer Suchmuster, d.h. Suchmuster die Elemente aus unterschiedlichen Bereichen des zu untersuchenden Quelltextes betrachten, wird durch die Modularisierung dieser Muster unterstützt. Die Vorteile der Modularisierung ergeben sich aus der besseren Übersichtlichkeit einzelner kleinerer Musterfragmente, der Möglichkeit von Konsistenzprüfungen bereits einzelner separater Muster sowie der Option Suchmuster arbeitsteilig zu erstellen.

Da es im Rahmen dieser Arbeit notwendig ist, die Detailebene der zu analysierenden Quelltexte zu betrachten, muss festgelegt werden, auf welche Programmiersprache sich die zu spezifizierenden Muster beziehen. Es wurde die objektorientierte Programmiersprache Java [GJS+05] gewählt, da Java in Programmierübungen an Universitäten stark verbreitet ist, so dass ein größtmögliches Einsatzgebiet des in dieser Arbeit entwickelten Ansatzes gewährleistet wird. Es wird im Folgenden vorausgesetzt, dass der Java Code syntaktisch entsprechend der Java 2 Grammatik korrekt ist. Die Repräsentation des Quelltextes und der hierauf aufsetzenden Spezifikationssprache JPL wurden so festgelegt, dass die Darstellung der Syntaxelemente eine mögliche Anpassung an weitere Programmiersprachen unterstützt (s. Kapitel 3.2).

Im Folgenden werden die Anforderungen und Herausforderungen an die Quelltextrepräsentation spezifiziert, auf der die JPL aufsetzt:

- Da die Muster sowohl die Detailebene des Codes, als auch größere Strukturen erfassen sollen, ist es notwendig, Spezifikationen auf verschiedenen Abstraktionsebenen zu ermöglichen.
- Eine möglichst vollständige Erfassung der Syntax der Sprache Java ist notwendig, um bei der Suchmusterdefinition auf Detailebene keinen Einschränkungen zu unterliegen.
- Um den Ansatz auf weitere objektorientierte Sprachen ausweiten zu können, muss ein möglichst großer Teil der Syntaxelemente sprachunabhängig definiert werden.
- Die Repräsentation des Quellcodes muss die Konsistenz zwischen dem originalen Quellcode und dessen Darstellung ermöglichen. So müssen alle Syntaxelemente des Quelltextes und deren Beziehungen abgebildet werden können.

Basierend auf den Anforderungen der Problemdomäne und der Quelltextdarstellung werden nun die Anforderungen an die Musterspezifikationssprache JPL formuliert. Einbezogen werden sowohl funktionale Aspekte, als auch Kriterien, welche eine gute Anwendbarkeit der Spezifikationssprache sicherstellen sollen:

- Die Muster müssen visuell erstellt werden können, da diese Darstellungsart als intuitiver für den Anwender angenommen wird, als eine rein textuelle Spezifikation. So können Beziehungen zwischen Elementen über Kanten direkt dargestellt werden und die Reihenfolge der zu suchenden Elemente muß nicht angegeben werden. Dem Problem, dass graphbasierte Suchmuster durch den Einsatz vieler Einzelelemente unübersichtlich werden, begegnet die Sprache mit Abstraktionsmöglichkeiten, und dem Ansatz Muster modular zu erstellen.
- Bereits durch die Spezifikationssprache muss eine stark strukturierte Vorgehensweise vorgegeben werden, um somit Fehlern vorzubeugen.
- Die Sprache muss eine formale Fundierung haben, um die Konsistenz der erstellten Suchmuster zu gewährleisten. So dürfen z.B. keine Kanten generiert werden die nur mit einem Knoten verbunden sind.
- Die Sprache muss skalieren, d.h. es müssen sowohl kleinere Muster als auch hierauf aufbauend komplexe Muster erstellt werden können.
- Die Sprache muss die Erfassung einer großen Anzahl an Implementierungsvarianten zu gegebenen Anforderungen unterstützen. Der Aufwand für die spezifizierende Person muss hier möglichst gering gehalten werden.

Zur praktischen Anwendung der Musterspezifikationssprache ist die konsistente Überführung der in JPL-Mustern enthaltenen abstrakten Elemente in konkrete JCG-Strukturen notwendig, welche nachfolgend automatisiert auf der JCG-Struktur der Quelltextrepräsentation der Übungsaufgabe gesucht werden. Da die JPL-Suchmuster bereits visuell erstellt werden, sollte zur Ableitung ebenfalls eine graphische

Repräsentation gewählt werden, um einen Systembruch zu vermeiden, und Konsistenzproblemen zwischen der Musterspezifikation und der Mustersuche vorzubeugen.

Zur Anwendung der Sprache ist eine durchgängige Methodik notwendig, welche Methoden zur Erstellung von Suchmustern und nachfolgend deren Spezifikation sowie die automatisierte Testausführung beschreibt. Dieser Prozess muss auf die hier betrachtete Domäne der Analyse von Übungsaufgaben ausgerichtet sein.

Im Folgenden wird der Beitrag der Arbeit zu den oben genannten Anforderungen zusammengefasst. Hierbei werden sowohl die konzeptionellen Herausforderungen der JPL, als auch die zur automatisierten Überprüfung notwendigen Tools dargestellt.

1.3. Beitrag der Arbeit

Es wird ein Ansatz zur Spezifikation von Suchmustern über Java Quellcode betrachtet, der auf einer graphbasierten Repräsentation sowohl des Quelltextes als auch der Suchmuster basiert. Folgende Forschungsfragen stehen im Fokus der Arbeit:

1) Wie kann ein Suchmuster über Quellcode strukturiert erstellt und übersichtlich dargestellt werden, das eine Vielzahl an Elementen und Beziehungen enthält?
2) Wie kann die Vielfalt möglicher Implementierungsvarianten zu einer Problemstellung in einem Suchmuster erfasst werden, ohne jede Variante detailliert beschreiben zu müssen, oder auf Grund einer zu hohen Abstraktionsebene *false positives* zu generieren?

Die Spezifikation der zu suchenden Muster erfolgt visuell über die *Java Pattern Language (JPL)*. Die Sprache realisiert ein Modulkonzept zur strukturierten Erstellung komplexer Muster. Dieses Konzept ist angelehnt an die Darstellung modularer Softwarearchitekturen der Spezifikationssprache II [GS94].

Die Syntax der JPL umfasst sowohl Elemente des *Java Code Graphen (JCG)* [Wie04] als auch des *Systemabhängigkeitsgraphen (SDG)* [HR92], welche im Rahmen dieser Arbeit erweitert werden. Weiterhin wird die JPL-Schnittstellenstruktur eingeführt, welche die Spezifikation von Datenübergabestrukturen in komplexen Mustern über mehrere Module hinweg unterstützt.

Der grundlegende Unterschied zwischen der JPL und den bestehenden Modularisierungsansätzen, welche in Kapitel 2 dargestellt werden, liegt im Detaillierungsgrad der Einheiten, welche durch die Module abgebildet werden. Während sich Module dieser Ansätze eher auf größere Einheiten wie Klassen oder Mengen von Klassen beziehen, wird in dieser Arbeit die Detailebene des Quelltextes zusätzlich mit einbezogen, so daß z.B. auch eine Schleife als Modul spezifiziert werden kann.

Ein wichtiges Konzept der JPL ist die Abbildung des Gültigkeitsbereichs von Variablen als modulare Einheit. Somit können sowohl große Strukturen wie Klassen und Pakete als auch detaillierte Strukturen, wie Methoden, Schleifenkörper oder Bedingungskörper einheitlich als Module abgebildet werden. Hierdurch wird das Muster, welches sonst im gesamten Quellcode gesucht würde, in einen engeren Kontext, nämlich den Kontext des Gültigkeitsbereichs, gesetzt und der zu durchsuchende Bereich somit eingeschränkt. So

wird u.a. bei einfacheren Codestrukturen vermieden, dass diese zwar im Quelltext gefunden werden, allerdings nicht an der erforderlichen Position.

Der Fokus der JPL liegt auf drei Aspekten:

- Der Ansatz ermöglicht es, komplexe Muster über einen modularen Ansatz strukturiert zu beschreiben. Die Verbindung der Module erfolgt über deren Schnittstellen.
- Die Spezifikation der Verbindung zwischen den Modulen kann unabhängig von deren Implementierung im zu untersuchenden Quelltext erfolgen. Somit können über eine abstrakt spezifizierte Modul-Kopplung alle möglichen Implementierungsvarianten zur Realisierung dieser Verbindung im zu analysierenden Quellcode automatisch erkannt werden.
- Im Suchmuster können sich Elemente verschiedener Abstraktionsebenen überlagern, so dass der Nutzer bestimmen kann, wie detailliert er sein Suchmuster definiert. Diese Kombinationsmöglichkeit ermöglicht ihm z.B. zu entscheiden, auf welche Weise er die Implementierungsvarianten zu einer Anforderung abbilden möchte.

Die Transformation der abstrakten in JPL formulierten Suchmuster zu konkreten Mustern, die im Quelltext gesucht werden können, erfolgt über die Technik der Graphtransformation [Roz97]. Hierbei werden die einzelnen Teile einer JPL-Spezifikation durch Graphtransformationsregeln abgeleitet. Über den sich ergebenden Regelsatz können nachfolgend automatisiert die Graphrepräsentationen der zu untersuchenden Quelltexte überprüft werden.

Die Technik der Graphtransformation wurde gewählt, da diese formal fundiert ist und somit die Konsistenz zwischen abstrakter Spezifikation und detaillierten ausführbaren Suchmustern gewährleistet werden kann. Zum anderen werden die Regeln der Graphtransformation visuell formuliert, so dass kein Modellbruch auf dem Ableitungsweg von der JPL-Musterspezifikation zu ausführbaren detaillierten Mustern entsteht.

Die Toolumgebung, welche für die Ausführung der Musterableitung und die Mustersuche verwendet wird, basiert auf dem AGG (Attributet Graph Grammar) [agg14] System, welches sowohl die Erstellung als auch die Ausführung von Graphtransformationsregeln unterstützt. Die Transformation eines JPL Musters in eine Menge von Graphregeln, welche zur automatisierten Suche über das AGG Tool ausgeführt werden können, wird in dieser Arbeit dargestellt.

Ergänzend werden Strategien zur Erstellung von Suchmustern für die Domäne der Unterstützung der Übungsaufgabenkorrektur [KG06] vorgestellt, so dass ein Nutzer der Musterspezifikationssprache einen methodischen Leitfaden für den Einsatz der JPL erhält. Damit werden für die JPL sowohl konzeptionelle als auch methodische und technische Aspekte beschrieben und realisiert.

Der in dieser Arbeit vorgestellte Ansatz erfüllt somit die unter 1.2 aufgeführten Anforderungen und hebt sich auf zwei Ebenen von bisher bestehenden Ansätzen ab:

- Quantitative Ebene:
    - Durch die Kapselung von Implementierungsvarianten in abstrakten JPL-Schnittstellenstrukturen wird der Spezifikationsaufwand für komplexe Muster gegenüber Techniken, die direkt auf dem AST (Abstract Syntax Tree) aufsetzen, verringert. Dies wird in der Bewertung der Anwendungsbeispiele in Kapitel 8 dargestellt.
- Qualitative Ebene:
    - Durch die Spezifikation von Suchmustern über eine Modulstruktur wird die strukturierte Erstellung von komplexen Mustern, die mehrere Gültigkeitsbereiche umfassen, unterstützt. Sowohl die modulare Darstellung in der JPL als auch die Beschreibung der Modulverbindungen werden in Kapitel 4, bzw. im Speziellen in den Kapiteln 4.3 und 4.4 erläutert.
    - Durch die Fokussierung der Module auf Gültigkeitsbereiche kann der Suchraum im Rahmen der Spezifikation auf diese eingeschränkt werden. Die Darstellung, wie Gültigkeitsbereiche in der JPL spezifiziert werden, erfolgt in Kapitel 4.3 durch die Beschreibung der Identifikator Sektion.
    - Durch die kombinierte Verwendung von Elementen verschiedener Abstraktionsebenen hat der Nutzer einen großen Freiheitsgrad in der Musterspezifikation und der Erfassung von Implementierungsvarianten. Die verschiedenen Elemente, welche zur Musterspezifikation verwendet werden können, werden in Kapitel 3 und Kapitel 4 dargestellt.
    - Durch zusätzliche Kontextinformationen zum gefundenen Muster wird die manuelle Analyse zur Vermeidung von „false positives“ unterstützt. Zu Beginn des Kapitels 4.4 wird anhand eines Beispiels der Mehrwert dieser Informationen (Angabe der Art der Variablenübergabe) dargestellt.

Im folgenden Kapitel wird der Aufbau der Arbeit näher erläutert.

1.4. Aufbau der Arbeit

Die Arbeit ist in 5 Hauptabschnitte untergliedert:

Im *ersten* Abschnitt wird die Repräsentation des Programmcodes als Graphstruktur dargestellt. Im *zweiten* Abschnitt werden sowohl die Konzeption der JPL als auch die Ableitung der abstrakten Suchmuster über Graphtransformationsregeln in detaillierte Strukturen, über die nachfolgend die Suche ausgeführt wird, erläutert. Im *dritten* Teil werden Methoden zur Anwendung der JPL besprochen und im *vierten* Teil wird die technische Realisierung zur Unterstützung der JPL beschrieben. Im *fünften* Teil wird die Sprache über verschiedene Anwendungsfälle evaluiert.

Kapitelübersicht im Einzelnen:

Kapitel 2 beginnt mit einem Vergleich von Techniken zur Mustererkennung, um darauf aufbauend den in dieser Arbeit gewählten graphorientierten Ansatz zu rechtfertigen. Nachfolgend werden Arbeiten vorgestellt, welche ebenfalls die Domäne der graphbasierten Mustererkennung auf Java Quellcode betrachten. Zuerst werden Arbeiten im Bereich der Analyse von Programmierübungen dargestellt. Nachfolgend werden Arbeiten beschrieben, welche die Bereiche der Design Pattern-Erkennung, des Refaktoring und der Identifikation von Services abdecken. Abschließend wird die Mustererkennung über die JPL zu diesen Ansätzen in Beziehung gesetzt und ihr Beitrag im Rahmen der Mustererkennungstechniken herausgearbeitet.

Kapitel 3 erläutert die graphbasierte Repräsentation des Quellcodes. Die Darstellung setzt auf den Strukturen des AST und SDG auf, so dass zuerst diese grundlegenden Strukturen beschrieben werden, um nachfolgend Anpassungen einzuführen, welche die Spezifikation von Suchmustern durch die JPL erleichtern.

Konzeption der JPL
Kapitel 4 stellt die modulare Struktur der JPL dar. Es wird sowohl auf das einzelne Modul unter ausführlicher Erläuterung der Schnittstellen als auch auf die Kopplung der Module zu komplexen Strukturen eingegangen. Besonders betrachtet werden hier die JPL spezifischen abstrakten JPL-Schnittstellenstrukturen, die zur Abstraktion von Implementierungsvarianten verwendet werden.

Kapitel 5 beschreibt die Ableitung eines in JPL spezifizierten abstrakten Musters durch Graphregeln. Im Weiteren wird dargestellt, wie mittels der hieraus resultierenden konkreten Muster die automatisierte Mustersuche durchgeführt wird.

Methoden zur Verwendung der JPL
Kapitel 6 zeigt Methoden auf, nach denen Suchmuster im Kontext der Domäne „Unterstützung der Korrektur von Übungsaufgaben“ erstellt werden können.

Toolumgebung
Kapitel 7 erläutert für die Domäne der Überprüfung von Übungslösungen die Automatisierung des Prozesses von der Spezifikation eines JPL-Musters bis zur Ausführung der Mustersuche. Es wird das Grobdesign eines Systems vorgestellt,

welches zur Unterstützung der Spezifikation und der automatisierten Ableitung von JPL Mustern verwendet werden kann. Weiterhin wird die bereits implementierte Systemumgebung zur automatisierten Prüfung von Übungslösungen vorgestellt, auf welcher das Grobdesign aufsetzt.

Anwendungsbeispiele und Fazit
Kapitel 8 evaluiert den Ansatz anhand verschiedener Beispiele, die auf Übungsaufgaben basieren, welche in der Veranstaltung „Grundlagen der Programmierung“ an der Universität Duisburg-Essen eingesetzt wurden.

Kapitel 9 fasst die Ergebnisse der Arbeit zusammen, zieht ein Fazit und beschreibt im Ausblick verschiedene Erweiterungsmöglichkeiten des hier präsentierten Ansatzes.

Im folgenden Kapitel werden verwandte Ansätze beleuchtet und die Sprache JPL in Relation zu diesen Ansätzen gesetzt.

2. Verwandte Arbeiten

Der statische Softwaretest umfasst im Rahmen der Quelltextanalyse verschiedene Methoden und Techniken, welche von der Suche nach Anomalien im Daten- und Kontrollfluss [HMR+03], der in dieser Arbeiten betrachteten Mustersuche, des Slicing (s. Kapitel 5.11), bis zu Methoden des Search Based Software Engineering [Har07] reichen, welche über Optimierungsalgorithmen z.B. die Domäne der Testdatenbestimmung betrachten [HM10]. Hinzu kommen metrikbasierte Verfahren zur Codeanalyse [Mat05], sowie Ansätze über genetische Algorithmen [Har07] oder Fuzzy Logik [NWW01]. In diesem Kapitel wird eine Abgrenzung der bestehenden Ansätze zu der vorliegenden Arbeit vorgenommen und abschließend der Mehrwert der JPL dargelegt.

Das folgende Kapitel beginnt mit der Erläuterung, warum eine graphbasierte Technik für die in dieser Arbeit betrachtete Problemstellung gewählt wurde. Mustererkennungstechniken im Rahmen des Tests von Studentenübungen werden in Kapitel 2.2. betrachtet, während der Bereich des Design Recovery in 2.3. und das Refaktoring in 2.4. beschrieben wird. Der Schwerpunkt der dargestellten Techniken liegt auf graphbasierten Verfahren, da diese den Fokus dieser Arbeit bilden. Alternative Techniken werden im Einzelfall zu den im Folgenden dargestellten verwandten Arbeiten hinzugenommen, um das in der jeweiligen Domäne verwendete Methodenspektrum aufzuzeigen.

Um umfangreiche Graphen erfassen zu können, werden die Techniken der Hierarchisierung und Modularisierung eingesetzt. In Kapitel 2.5. werden verschiedene Ansätze erläutert, welche im Kontext der Graphtransformation unterschiedliche Modularisierungstechniken betrachten. Nachfolgend wird der Modularisierungsansatz der JPL von diesen abgegrenzt.

Abschließend werden in Kapitel 2.6. die bestehenden Arbeiten bewertet, hinsichtlich ihrer Stärken und Schwächen eingeordnet und aufgezeigt, welche Lücken der Ansätze durch die hier vorgestellte Arbeit geschlossen werden.

2.1. Techniken zur automatisierten Mustererkennung im Quellcode

In diesem Kapitel werden verschiedene Techniken zur Analyse von Quelltextdateien mittels automatisierten Mustererkennungstechniken dargestellt und darauf aufbauend wird begründet, warum der graphbasierte Ansatz für diese Arbeit gewählt wurde.

Eine weit verbreitete Technik zur Textanalyse ist die Suche nach Mustern über reguläre Ausdrücke. Dieser Ansatz wird realisiert über die Nutzung des von UNIX unterstützten, „grep“ Befehls, mittels dessen nach regulären Ausdrücken in Textdateien gesucht werden kann [KP84]. Verschiedene Tools erweitern den grep Befehl dahingehend, dass auch komplexe Muster in Texten effizient gesucht werden können. Diese vom „grep“ Befehl abgeleiteten Techniken wurden in [PP94] um Formulierungsmöglichkeiten ergänzt, welche es erlauben, Mengen von Ausdrücken ohne eine vorgegebene Reihenfolge zu suchen. Weiterhin wurden die Möglichkeiten Querverweise zwischen Ausdrücken zu erstellen ausgeweitet, so dass es mit dem Einsatz dieses Tools erstmals möglich ist effiziente Abfragen über Muster im Quellcode zu erstellen. Die aktuell

prominentesten Vertreter sind agrep und nr-grep, die es zusätzlich ermöglichen Muster zu finden, welche dem Suchmuster nicht exakt entsprechen (approximative Mustersuche) [Nav01]. Außer den zuvor erwähnten Vertretern der „grep-Familie“ existieren noch zahlreiche weitere Derivate, wie z.B. fgrep, welches zur einfachen Patternsuche ohne reguläre Ausdrücke verwendet wird, oder (Web)Glimpse, welches Webseiten durchsucht.

Weitere auf regulären Ausdrücken basierende Techniken umfassen z.B. die Sprache Spine, welche zur Suche nach Design Pattern verwendet wird und direkt auf dem Quelltext aufsetzt [BBS05]. Unterstützt wird die Suche nach regulären Ausdrücken über verschiedene Methoden, welche den Suchraum eingrenzen [AFC98]. Aktuelle Techniken zur Mustererkennung setzen in der Mehrzahl nicht mehr direkt auf dem Quelltext auf, sondern bedienen sich einer Abstraktionsschicht, über welche die Mustersuche ausgeführt wird.

Über die Umwandlung von Quelltext in das Format XML (Extensible Markup Language) [XML14] wird es möglich Abfragen zu erstellen, welche die Struktur eines Dokuments, bzw. des hiermit dargestellten Quelltexts, mit einbeziehen. Zur Transformation des Source Codes wird die Transformationssprache XSLT (Extensible Stylesheet Language Transformation) zusammen mit XPath (XML Path Language) eingesetzt. Die hieraus entstehenden XML-Dateien, basieren auf einer Dokumenttypdefinition (DTD), in der die Typen aller (Quellcode)elemente definiert werden. Durch diese Abstraktionsschicht ist es möglich auf Typebene Abfragen zu stellen, wie z.B. die Suche nach Variablen, Schleifen, oder einer Methode. Allerdings sind größere XML-Dateien im Quelltext sehr unübersichtlich, so dass zur Darstellung des Inhalts spezielle graphische Oberflächen verwendet werden.

Weitere Ansätze nutzen Eigenheiten von Programmiersprachen, um die Mustersuche zu realisieren. So werden in [KP96] Ausdrücke der Prädikatenlogik in Prolog zur Musterspezifikation verwendet. Diese dienen der Ermittlung von Design Pattern in C++ Applikationen. Hierbei werden die zu suchenden Muster als Prolog-Regeln formuliert und für die Mustererkennung wichtige Teile des C++ Quelltexts (insb. die header-Dateien) in Prolog-Facts transformiert.

<u>Vorteile der graphbasierten Mustersuche gegenüber den zuvor besprochenen textbasierten Techniken:</u>

- Ein Suchmuster kann als Graphstruktur visuell gut beschrieben werden. Die Darstellung großer Graphen kann allerdings schnell unübersichtlich werden. Um diesem Problem zu begegnen liegt ein Schwerpunkt der JPL auf der modularen Repräsentation des Suchmusters. Auf diese Weise kann das Muster aus einzelnen Modulen strukturiert zusammengesetzt werden. Weiterhin können zur Spezifikation der Suchmuster Elemente verschiedener Abstraktionsebenen verwendet werden, um eine zu großen Menge von Elementen im Suchmuster durch Abstrahierung zu vermeiden.
- Beziehungen zwischen Elementen können graphisch über Kanten direkter formuliert werden als z.B. über reguläre Ausdrücke. Die in dieser Arbeit betrachteten Suchmuster (Muster zur Analyse von Übungsaufgaben) enthalten typischerweise eine Vielzahl dieser Beziehungen.

- Falls ein Suchmuster mehrere Teilmuster umfasst (z.B. die Beschreibung zweier Schleifen, in denen eine Methode aufgerufen wird), so muss die Reihenfolge, in der diese Muster auftreten sollen, nicht festgelegt werden, während textuelle und XML-basierte Ansätze hier die Angabe einer Reihenfolge erfordern.
- Die Problematik der doppelten Benennung (zwei Variable in unterschiedlichen Gültigkeitsbereichen besitzen den gleichen Namen und können bei Betrachtung des reinen Quelltexts somit nicht unterschieden werden) kann durch die Darstellung von Codeelementen als eigenständig identifizierbare Einheiten gelöst werden.
- Transformationen, welche in dieser Arbeit z.B. im Rahmen der Ableitung des abstrakten JPL-Suchmusters und für die Markierung von Teilmustern bei der sektionsbasierten Mustersuche notwendig sind, können über Graphtransformationen einfacher beschrieben werden, als unter Nutzung von XML-basierten Techniken, welche auf die generelle Transformation von XML-Strukturen ausgelegt sind.

Aus den oben genannten Gründen wird im Folgenden die Mustersuche über Graphstrukturen betrachtet. Hierbei wird in Kauf genommen, dass Graphstrukturen über den Quelltext erzeugt werden müssen, die für die zuvor genannten stringbasierten Techniken nicht notwendig sind. Da große Graphstrukturen mit vielen Elementen unübersichtlich werden können, wird in dieser Arbeit ein Fokus auf die modulare Unterteilung des Gesamtgraphen gelegt, so dass dieser verständlich bleibt. In den folgenden Kapiteln werden Ansätze aufgeführt, welche Muster im Quelltext in verschiedenen Kontexten identifizieren, und somit helfen den Quellcode zu analysieren und zu verstehen.

2.2. Analyse von Programmierübungen

In diesem Kapitel werden Arbeiten betrachtet, welche sich mit der automatischen Analyse von Programmierübungen über Musterlösungen befassen. Es werden die Bereiche der Überprüfung von Übungslösungen und der Plagiaterkennung unterschieden.

Programmanalyse
Die Idee Programmierübungen über die Suche nach Musterlösungen für Teile der Gesamtlösung zu analysieren, wurde in [TRB04] und [TBR06] als Teil eines kombinierten Ansatzes aus statischen und dynamischen Tests auf Übungsaufgaben konzipiert und realisiert. Die Anwendung der statischen Tests beschränkt sich in diesen Arbeiten auf Strukturen, welche ausschließlich auf Anweisungstypen basieren. So wird z.B. nicht zwischen verschiedenen Schleifenarten unterschieden und vom Inhalt der Ausdrücke wird vollständig abstrahiert. Die Suchmuster werden in einem XML-Format definiert, welches u.a. folgende Elemente umfasst:

<assignment>, <methodcall>, <method>, <loop>, <condition>, <trueBranch> , <falseBranch>, <statement>

Beispiel zur Umwandlung von Quellcode in XML:

Quellcode:

```
k=x;
l();
while(k==x){g();}
...
```

XML:

```
<assignment>k</assignment>
<methodCall>l</methodCall>
<loop>
        <condition>
                <trueBranch>
                        <methodCall>g<methodCall>
                </trueBranch>
        </condition>
<statements>
……
```

Aus diesen typisierten Beschreibungen der einzelnen Elemente im Suchmuster resultiert bei der Anwendung auf Übungsaufgaben eine hohe Anzahl von *false positives* (falsche Lösungen, welche als korrekt identifiziert werden), wobei die strikte Vorgabe der Reihenfolge der einzelnen Anweisungen die Anzahl der erfassten Programmvarianten stark einschränkt. Die Ergebnisse der Mustersuche werden in [TBR06] demzufolge nur als grobe Hinweise für mögliche Fehler in Übungsaufgaben gewertet. Die Betrachtung komplexer Strukturen ist aufgrund der erläuterten Einschränkungen kaum möglich.

Mustererkennungstechniken, welche direkt auf dem AST aufsetzen, werden in [Jac97] und [HMR+03] zur Überprüfung von Lösungen von Programmieraufgaben eingesetzt. Hierbei werden allerdings nur Muster gesucht, welche die Einhaltung eines vorgegebenen Programmierstils verletzen. Die Beschreibung des Programmierstils enthält u.a. folgende Anweisungen: Variablen werden immer kleingeschrieben, Ausdrücke in Schleifenrümpfen werden immer mit geschweiften Klammern umrahmt, etc. Weiterhin wird nach Strukturen gesucht, welche „typische Anfängerfehler“ finden sollen, wie z.B. die Verwendung des Operators = anstatt == in einer Bedingung, zwei Belegungen einer Variablen, ohne diese dazwischen auszulesen, etc. Dieser Fehlertyp wird Datenflussfehler genannt. Typische Fehler im Kontrollfluss umfassen z.B. nicht ausführbare Anweisungen, oder mehrere Eingänge oder Ausgänge aus Schleifen. Diese Strukturen können über Muster gesucht werden, ohne auf den Kontext der Aufgabenstellung eingehen zu müssen. Kontextabhängige Analysen sind in diesen Arbeiten nicht vorgesehen.

Plagiaterkennung

Die Plagiaterkennung wird in diesem Abschnitt betrachtet, da bei der in dieser Arbeit angestrebten Mustererkennung in Aufgabenlösungen „Plagiate“, d.h. von einer Musterlösung abweichende Lösungen mit identischer Funktionalität, als korrekt erkannt werden müssen. Hierbei kann auf die Techniken der Plagiatserkennung zurückgegriffen werden.

Ein Plagiat ist gekennzeichnet durch die Kopie einer existierenden Arbeit und der nachfolgenden Abwandlung einzelner Teile, so dass die Kopie als solche nicht ersichtlich ist. Die Erkennung von Plagiaten basiert auf der Suche nach Programmmustern, welche eine erhebliche Ähnlichkeit zu einer bereits existierenden Lösung aufweisen. Man spricht hier von der Erkennung von Kernteil-Plagiaten, welche zuvor festgelegt werden.

Es ist zu beachten, daß der Anwendungskontext der JPL und der Plagiatserkennung unterschiedlich ist. Während durch die JPL ein Muster vom Nutzer spezifiziert wird welches gesucht werden soll, wird bei der Plagiatserkennung sowohl der Originalcode, als auch der zu überprüfende Code automatisch auf ähnliche Codefragmente hin durchsucht. Dies bedeutet es ist hier nicht im Voraus bekannt welche konkreten Muster als Plagiat gesucht werden sollen. Der Nutzer gibt also keine Muster vor, sondern es werden Muster gesucht, welche auf Grund vorgegebener Kriterien als ähnlich definiert werden. Abgrenzend hierzu werden in der JPL individuelle Muster spezifiziert, wobei die zu suchenden Elemente durch den Nutzer festgelegt werden.

Folgende grundlegende Techniken zur Plagiaterkennung können nach [LCH+06] unterschieden werden:

- String basierte Algorithmen: Programme werden als Sequenzen von Strings betrachtet. Der Nachteil dieser Technik ist, dass zwischen Original und Plagiat eine 1:1 Beziehung bestehen muss. Bereits kleinere Unterschiede machen die Erkennung unmöglich.
- Token basierte Algorithmen: Identifier und Schlüsselwörter werden als Token repräsentiert. Das Programm wird als Sequenz von Token dargestellt. Der Prüfalgorithmus ist allerdings weiterhin anfällig für Änderungen der Anweisungsreihenfolge und das Einfügen von zusätzlichem Code.
- AST-basierte Algorithmen: Programme werden als AST betrachtet (für genauere Informationen zum AST sh. Kapitel 3.1). Da die einzelnen Anweisungen im AST explizit ausformuliert sind, können bereits Plagiate, welche auf der Änderung der Elementreihenfolge innerhalb einer Anweisung basieren, nicht mehr direkt identifiziert werden. Weiterhin wird bei direkter Verwendung des AST die Anweisungsreihenfolge vorgegeben, so dass Änderungen in dieser Sequenz nicht mehr als Plagiat erkannt werden.
- PDG (Program Dependency Graph) basierte Algorithmen: Durch diese Technik wird der Daten- und Kontrollfluss eines Programms analysiert. Plagiate, welche sich durch zusätzlichen Code oder die Änderung der Anweisungsreihenfolge vom Quelltext unterscheiden, werden nun erkannt. Aus der ausschließlichen Verwendung dieser Abstraktionsebene resultiert allerdings eine große Anzahl von „false positives“, durch welche diese Art der Plagiatserkennung als sehr ungenau einzustufen ist.

Im Unterschied zu den gennannten Techniken können in der JPL Syntaxelemente, welche auf dem AST und PDG basieren, gemeinsam in der Suchmusterspezifikation verwendet werden, um somit die Zuverlässigkeit der Mustersuche zu erhöhen. Die Einführung des Elements *Pfad* ermöglicht es, Sequenzen von Ausdrücken über den AST zu spezifizieren, ohne dass zwischenliegender Quellcode die Mustersuche beeinträchtigt. Weiterhin wird in der JPL das Suchmuster modular aufgebaut, so dass bei großen Strukturen die Übersichtlichketi erhalten bleibt.

2.3. Suche nach Pattern im Quellcode zum Programmverstehen

Ein in der Literatur umfangreich beschriebenes Themenfeld im Rahmen der Mustererkennung im Quellcode, umfasst die Suche nach Design Pattern, um so Teilbereiche aus einer komplexen Codestruktur zu isolieren und darauf aufbauend den Quelltext besser verstehen zu können [DZP09].

Die ersten Verfahren zur Design Pattern Erkennung basierten auf der Angabe von Zusatzinformationen im Quelltext. Somit kann, basierend auf diesen Angaben, welche entweder als zusätzliche „Strukturdatei“, oder direkt im Quellcode aufgeführt werden, auf Muster im Code zurückgeschlossen werden. Beispiele für C++ und Java werden in [KSR+99], gestützt durch entsprechende Tools zur automatischen Auswertung der strukturellen Informationen, erläutert. Da diese Zusatzinformationen in realistischen Projekten aus der Praxis nicht gegeben sind, und die in dieser Arbeit betrachteten Übungslösungen von Studenten diese ebenfalls nicht enthalten, werden im Folgenden nur Arbeiten betrachtet, welche ausschließlich auf dem Quellcode aufsetzen.

Graphbasierte Techniken

Analyse über Klassenstrukturen

Erste Arbeiten, welche keine Zusatzinformationen im Code benötigen, realisieren die Design-Pattern Suche über die Analyse der Klassenstrukturen. Hierbei liegt der Fokus auf Vererbungs- und Aufrufbeziehungen zwischen Klassen, welche z.B. für UML Diagramme in [KP96], [BBC08] und über den AST der Sprache C++ in [AFC98] betrachtet werden.

Analyse über Prädikatenlogik

Die Graphabfragesprache GReQL [GRe14] basiert auf Prädikatenlogik und realisiet durch reguläre Ausdrücke Abfragen über Graphstrukturen. Die elementaren Sprachelemente sind FWR-Ausdrücke (From With Report -> Ausgangselemente, zu suchender Ausdruck, Ergebnisreport), quantifizierte Ausdrücke, reguläre Pfadausdrücke und Funktionsanwendungen. Im Fokus stehen hier Abfragen über Java Quellcode, welcher als TGraph (typisierter**,** attributierter**,** angeordneter und gerichteter Graph) repräsentiert wird. In [EB10] werden Einsatzbeispiele im Rahmen der Quellcodeanalyse gegeben. So eignet sich die Sprache besonders gut zur Erreichbarkeitsanalyse und damit der Analyse von Beziehungen. Somit können z.B. Methodenaufrufe, die Analyse vorgegebener Anweisungsabläufe und die Ermittlung von Metriken durch Ausdrücke in GReQL dargestellt werden.

Analyse über UML und Quelltext

Die meisten Techniken, welche den vollständigen Quelltext in die Suche nach Design-Pattern [GHJ+95] mit einbeziehen, basieren auf dessen Darstellung im AST. Für die Analyse werden mögliche Implementierungsvarianten als Graphstruktur erstellt und diese verschiedenen Strukturen auf der AST-Struktur des zu analysierenden Quellcodes gesucht. Größere Suchmuster werden über diese Techniken allerdings schnell zu komplex, und sind ohne weitere Abstraktionen nicht mehr handhabbar.

In [NSW+02] und [NMW04] wird ein graphtransformationsbasiertes Verfahren zur Design Pattern Erkennung vorgestellt, welches zur Identifikation einen hierarchischen Aufbau mehrerer voneinander abhängiger Suchmuster nutzt. Hierbei werden zuerst

kleinere Sub-Pattern im Quelltext gesucht und über Annotationen als Bestandteile von Design Pattern gekennzeichnet. Aufbauend auf diesen Annotationen werden nachfolgend abstraktere Patternstrukturen gesucht, so dass nach mehreren Iterationen auch umfangreiche Design Pattern erkannt werden können, welche sich über verschiedene Methoden und/oder Klassen erstrecken können. Die inkrementelle Analyse des AST erfolgt halbautomatisch, da Beziehungen zwischen Sub-Pattern, welche für ein Design Pattern notwendig sind, zuvor vom Designer der Suchmuster festgelegt werden müssen. Implementierungsvarianten werden durch verschiedene Ausprägungsmöglichkeiten eines Sub-Pattern beschrieben, welche in einem Patternkatalog erfasst werden. Weiterhin können strukturelle Komponenten als optional gekennzeichnet werden, wobei gefundene Muster höher bewertet werden, wenn diese enthalten sind. Der Ansatz wird in der Fujaba Tool Suite [Fuj14] realisiert, welche sowohl die Systemmodellierung mittels UML Diagrammen und Story Diagrammen ermöglicht, als auch die automatische Generierung von normalisierten AST Strukturen aus Java Quellcode unterstützt. Die Mustererkennung erfolgt auf Grundlage des zuvor beschriebenen Prozesses und wird durch die Tool Suite Reclipse realisiert, welche auf FuJaba aufsetzt [DMT10]. Die Suchmuster für die Sub-Pattern und die Design Pattern werden in einer UML-ähnlichen Spezifikationssprache beschrieben und in einem Musterkatalog hinterlegt. Nachdem ein Design Pattern erkannt wurde, wird die Qualität der gefundenen Struktur bewertet, d.h. es wird abgeschätzt, wie wahrscheinlich es ist, dass das zu suchende Design-Pattern gefunden wurde, oder ob ein „false positive“ vorliegt [Tra06]. Hierbei wird prozentual bewertet, wie groß die Abdeckung zum gesuchten Pattern ist, z.B. wie viele und welche optionalen Attributwerte der UML-Elemente gefunden wurden. Zur nachfolgenden manuellen Analyse können die gefundenen Matches angezeigt werden.

Die in [NSW+02] dargestellte Patterndefinitionssprache setzt auf der UML Spezifikation und dem Abstrakten Semantischen Graph, einem normalisierten AST, auf und ähnelt der in dieser Arbeit verwendeten Mustersuche über die Java Code Language (s. Kapitel 4). Allerdings unterstützt diese weder ein modulares Konzept, welches Modulschnittstellen einbezieht, noch werden Strukturen zur Daten- oder Kontrollabhängigkeit betrachtet.

In [SO06] wird die Suche nach Design Pattern über UML-Derivate, den AST und den Programmfluss durchgeführt. Hierbei wird initial der Suchraum über die Identifikation von Klassen, die ein Pattern realisieren könnten, eingegrenzt, und nachfolgend innerhalb der Methoden dieser Klassen über den AST, sowie den Kontroll- und Datenfluss nach Detailstrukturen zu diesen Pattern gesucht. Die Suchalgorithmen zu einzelnen Pattern sind fest codiert, so dass die freie Spezifikation eines Suchmusters nicht möglich ist. Realisiert wird dieser Ansatz über das Tool PINOT. Ergänzend hierzu wird in [Nij07] der graphische Editor MUSCAT vorgestellt, über den die Pattern auf UML-Ebene instanziiert werden können, d.h. es können Klassen und deren Beziehungen untereinander angegeben werden. Über diese Klassen und Beziehungen können nachfolgend sogenannte *Pattern Rules* ausgeführt werden. Diese Regeln suchen nach festgelegten Detailmustern, welche den verschiedenen Design Pattern entsprechen. In [Nij07] werden 13 dieser Regeln vorgestellt.

Weitere Techniken

In [KB00] und [FWM+96] werden Design Pattern über Metriken charakterisiert. Hierbei werden objektorientierte Metriken (Methoden per Klasse, Vererbungstiefe, etc.), strukturelle Metriken (Fan-In/Fan-Out, Strukturelle Komplexität) und Metriken auf Methodenebene (Lines of Code, McCabes Zyklomatische Kompexität, etc.) verwendet. Die über diese Metriken gefundenen Strukturen beziehen sich hauptsächlich auf Pattern, welche auf Vererbungsbeziehungen und Assoziationen beruhen. Die korrekte Erkennung von detaillierten Mustern ist allerdings sehr ungenau, so dass hier eher das allgemeine Codeverständnis im Vordergrund steht und die Technik für den in dieser Arbeit betrachteten Bereich nicht direkt verwendet werden kann. In [TDB11] wird ein Ansatz beschrieben, welcher die Programmanalyse über Metriken (clustering based analysis), mit der zuvor beschriebenen Muster-basierten Analyse (Pattern based analysis) kombiniert. Die Idee ist, den Suchraum initial über den Einsatz von Metriken einzuschränken (z.B. im Rahmen der Clusteranalyse die Betrachtung von Komponenten, die eine starke Kopplung besitzen) und nachfolgend detailliertere Muster über diese Elemente zu suchen. Diese Muster können einen starken Einfluss auf die zuvor eingesetzten Metriken ausüben und somit wäre es z.B. möglich Antipattern zu identifiziert (z.B. Ermittlung unnötig eng gekoppelte Komponenten).

In [Pet05] wird über eine Analyse des dynamischen Verhaltens ermittelt, an welcher Stelle erhöhter Aufwand bei der Strukturanalyse betrieben werden sollte. In [NWW01] werden Fuzzy Logic Algorithmen verwendet, um Muster von Design Pattern über Ähnlichkeiten zu identifizieren. Hierbei werden Strukturen im Quelltext mit in einem Design Pattern vordefinierten Mustern verglichen.

Zusammenfassend können die oben betrachteten Arbeiten unterteilt werden in Ansätze, die ausschließlich auf die Suche nach Design Pattern ausgerichtet sind (z.B. Muscat [Nij07]), und Ansätze, über die generische Suchmuster erstellt werden können (z.B. Reclipse [DMT10]). Der Fokus der Ansätze, über die allgemeine Muster spezifiziert werden können, liegt in der Analyse der Beziehungsstrukturen auf Klassen- und Methodenebene. Ausnahmen bilden FuJaba [Fuj14], welches die Spezifikation innerhalb einer Methode liegender Strukturen graphbasiert unterstützt, und GReQL [GRe14], in dem Beziehungen innerhalb von Methoden über Ausdrücke der Prädikatenlogik gesucht werden können. Der Unterschied der hier vorgestellten Arbeit zu diesen Ansätzen liegt in der modularisierten Suchmusterdarstellung, als auch in den Abstraktionstechniken, über die Quelltextstrukturen spezifiziert werden können.

Im Folgenden wird der Einsatz der Muster-Analyse in der Domäne des Refaktorings von Quelltext betrachtet. Neben der Design-Pattern-Suche liegt hier der zweite Schwerpunkt graphtransformationsbasierter Techniken zur Mustererkennung.

2.4. Suche nach Programmstrukturen zum Refaktoring

Ein Refaktoring ist definiert als *funktionserhaltende strukturelle Veränderung von Quellcode* [Fow99] mit dem Ziel der besseren Verstehbarkeit und damit Wartbarkeit von Softwareartefakten. Im Folgenden werden Techniken betrachtet, welche geeignete Strukturen im Quellcode suchen, um diese durch den Einsatz eines Refaktorings zu verändern.

Techniken zur Mustersuche im Quelltext über Graphtransformation

Um die Funktionserhaltung während der Modifikation des Codes sicherzustellen, werden Refaktorings als Programmtransformationen spezifiziert, welche gegebene Vor- und Nachbedingungen einhalten müssen. Da sich für die Graphtransformation bereits formal fundierte Techniken zur Konsistenz- und Bedingungsüberprüfung etabliert haben [BPT02] [Rij97] [MDJ02] und die visuelle Spezifikation von zu transformierenden Strukturen die Erkennung von Inkonsistenzen für einfache Strukturen erleichtert [MT04], werden graphtransformationsbasierte Verfahren häufig sowohl zur Erkennung der zu verändernden Struktur als auch der eigentlichen Veränderung eingesetzt.

Der am weitesten verbreitete Ansatz zur statischen Mustererkennung von Stellen im Quellcode, die nachfolgend durch ein Refaktoring verändert werden können, basiert auf „bad smells“, also Softwarestrukturen, welche die Vermutung nahe legen, dass hier unstrukturierter Code vorliegt, der schwer wartbar und somit für Refaktorings geeignet ist. In [BYM+98] und [BMD+00] werden Ansätze vorgeschlagen, welche doppelte Code-Strukturen eliminieren, indem auf einem AST-Graphmodell des Quellcodes entsprechende Muster gesucht und Transformationen ausgeführt werden, welche entweder Code aus Methoden extrahieren oder doppelte Methoden löschen. Die Suche nach doppeltem Code ist ein komplexes Gebiet in der Domäne der Refaktoringforschung, da der Terminus “doppelt” sich auf die Funktionalität und nicht auf die Syntax des Codes bezieht. Weitere Ansätze sind in [CDF+03] realisiert worden, welche auf der Suche nach häufig vorkommenden ähnlichen Pattern basiert. In [KH01] und [Kri01] wird zur Mustersuche der Systemabhängigkeitsgraph betrachtet.

In [BPT04] werden sowohl der AST des jeweiligen Softwarefragments, als auch dessen UML-Darstellung in die Untersuchung mit einbezogen werden. Die Arbeit basiert auf Beziehungen zwischen AST-Elementen und UML-Elementen, welche über Graphmappings realisiert werden. Die Kombination der beiden Abstraktionsebenen ermöglicht die Spezifikation von Mustern, welche mehrere Implementierungsvarianten beinhalten. Der Fokus dieses Ansatzes liegt allerdings nicht auf der erweiterten Mustersuche für Refaktorings, sondern der kohärenten Veränderung des Codes und gleichzeitig dessen Dokumentation in UML. Im Gegensatz zur JPL kann in [BPT04] nicht von der Art der Variablenübergabe zwischen Gültigkeitsbereichen abstrahiert werden, der Systemabhängigkeitsgraph wird für die Quellcodespezifikation nicht betrachtet und es erfolgt keine Modularisierung welche explizit den Export und Import der verschiedenen Bereichen graphisch als Modulsektion erfaßt.

In [MTR06] wird die Technik der „critical pair analysis“, d.h. eine Analyse der Abhängigkeiten zwischen Refaktorings verwendet. Die Ergebnisse bieten eine Hilfestellung zur Ermittlung, welche Refaktorings in einem speziellen Kontext (gegebenem Quellcode) einsetzbar sind. In [HJE+06] wird darauf hingewiesen, dass die traditionelle Graphtransformationstechnik nicht ausreicht, um komplexe Refaktorings zu beschreiben. Als erster Ansatz wird vorgeschlagen die Graphtransformation um die Technik des *Cloning* zu erweitern.

Aktuelle Arbeiten im Umfeld des Refaktorings über die Technik der Graphtransformation betrachten die Herausforderung des Refaktorings in umfangreichen Applikationen im Rahmen des „Search Based Engineering“ Ansatzes, d.h. sie führen die Mustersuche und nachfolgende Ausführung des Refaktorings auf ein Optimierungsproblem zurück. So werden Abhängigkeiten zwischen den für das

Refaktoring notwendigen Transformationsregeln über Techniken der Graphtransformation erkannt, woraufhin die zu bestimmende Abfolge der Regeldurchführungen als Optimierungsproblem formuliert wird [Qay10].

Graphbasierte Techniken zur Mustersuche über Metriken

Von Dudziak and Wloka [DW02] wird ein Ansatz vorgeschlagen, welcher auf Heuristiken basiert, die den entsprechenden „bad smell"-Mustern zugeordnet werden. Grundlage zur Ermittlung der Heuristiken bildet der AST. Beispiele sind hier z.B. Lazy Class für eine Klasse, welche kaum von anderen genutzt wird und Large Class, welche entweder zu viele oder zu große Methoden besitzt. Die Problematik dieses Ansatzes liegt in der Quantifizierung des Ausdrucks „zu viel". Objektorientierte Metriken werden in [SSL01] betrachtet, wobei Metriken über Methoden und Membervariablen im Vordergrund stehen. Die Technik basiert auf der Kohesionsanalyse von Methoden, aus der entsprechende Empfehlungen für Refaktorings abgeleitet werden. Weitere Arbeiten [MA10] über Metriken suchen nach Mustern, die sich sowohl auf Methodenaufrufe als auch auf Variableneigenschaften (static, private, etc.) beziehen. Das Ziel der Patternanalyse ist hier die Evaluation der Software Qualität.

In diesem Kapitel wurden Ansätze vorgestellt, die initial Muster in Graphstrukturen suchen, um diese nachfolgend über Transformationsschritte im Rahmen von Refaktorings zu verändern. Die hier dargestellte Suche nach „bad smells" über Graphtransformationstechniken wird in dieser Arbeit aufgenommen, und um Techniken der Unterteilung und Abstraktion der Muster ergänzt.

2.5. Hierarchische und modulbasierte Ansätze zur Graphtransformation

Die aufwendige Handhabung großer Graphen, und die hieraus folgende Notwendigkeit zur Abstraktion und Modularisierung, wurden bereits in verschiedenen Arbeiten betrachtet.

Im Kontext der Graphtransformationstechnik ist der hierarchische Ansatz aus [GTS01] zu nennen, welcher einzelne abstrakte Knoten über mehrere hierarchische Ebenen hinweg verfeinert, indem die Details in Graphstrukturen beschrieben werden, welche den Knoten zugeordnet sind. Beispielhaft wird eine Benutzerschnittstelle dargestellt, deren Elemente erst allgemein (abstrakter Knoten -> allgemeiner Typ einer graphischen Komponente) und nachfolgend in ihren Details (abgeleiteter Graph -> Ausprägungen und weitere graphischen Unterelemente der Komponente) betrachtet werden.

In [BEM+00] wird eine Verallgemeinerung des hierarchischen Ansatzes über „graph packages" angestrebt. Die grundlegende Idee ist hier die Spezifikation der Hierarchie initial vom Graph zu entkoppeln und die Graphelemente nachfolgend den entsprechenden Stufen zuzuordnen. Die Sichtbarkeit, und damit die Möglichkeit des Zugriffs auf Knoten verschiedener Packages, wird über Import- und Export–Schnittstellen realisiert. In [BKK05] wird dieser Ansatz aufgegriffen, weitergeführt und der Fokus auf die regelbasierte Veränderung dieser hierarchischen Strukturen gelegt. Es wird gezeigt, dass über die Graphtransformationstechnik des Double-Pushout-Approaches [CMR+97] die hier spezifizierten hierarchischen Strukturen regelbasiert

erfasst und z.B. in eine einfache Struktur, welche nur eine hierarchische Ebene benötigt, übertragen werden können.

Die zuvor beschriebenen Ansätze ermöglichen eine unbeschränkte Anzahl an zu spezifizierenden Hierarchieebenen. Eine Methode, welche hierarchische Strukturen mit einem modularen Ansatz vereint, wird in [GPS+99], [GEM+99] vorgestellt Die *verteilte Graphtransformation*, welche hierarchisch zwischen dem Netzwerkgraphen mit seinen Netzwerkknoten und Kanten sowie den diesen Knoten zugeordneten lokalen Graphen unterscheidet, ist zwar auf diese beiden Ebenen limitiert, führt aber im Gegensatz zum vorhergehenden Ansatz die Verbindung der lokalen Graphen mittels Export- und Importschnittstelle ein und legt für hierauf auszuführende Transformationen Bedingungen fest. Um die Konsistenz des verteilten Graphen nach einer Transformation sicherzustellen, müssen diese Bedingungen, welche sich sowohl auf die Netzwerkebene als auch die lokalen Graphen erstrecken, von den Graphtransformationen auf den entsprechenden Ebenen eingehalten werden.

Eine Anwendung dieser Technik wird in [Mey00] im Rahmen der Beschreibung von Services einer Softwarekomponente dargestellt. Eine Servicebeschreibung basiert häufig auf Informationen, welche als Kommentar zu dem Service hinzugefügt werden (ähnlich der manuellen Annotationen zur Beschreibung von Design Pattern). In [Mey00] wird ein Ansatz vorgestellt, welcher eine Servicebeschreibung über Graphmuster propagiert. Die Erstellung dieser Muster sowie die Suche nach den korrekten Mustern, werden über die Technik der verteilten Graphtransformation realisiert. Die Muster besitzen den Anspruch einer semantisch reichhaltigen Beschreibung eines Services, so dass diese auf einer höheren Abstraktionsebene als dem Quellcode formuliert werden müssen. In [Mey01] wird nicht vorgegeben, über welches Modell die Musterbeschreibung erfolgen soll. Beispiele werden im Zusammenhang mit UML-Diagrammen (Zustandsdiagramme, Klassendiagramme) und Petri-Netzen gegeben, wobei offen gelassen wird, ob diese eine „optimale“ Beschreibung der Komponente ermöglichen oder ob evtl. alternative Beschreibungen z.B. mittels Architekturbeschreibungssprachen besser wären. Diese Entscheidung wird dem Nutzer überlassen, welche dieser kontextabhängig treffen muss.

Abgrenzung: Eine Musterdarstellung über Hierarchieebenen steht in dieser Arbeit nicht im Fokus, jedoch könnte die modulare Erfassung der verschiedenen Gültigkeitsbereiche als implizite Hierarchie verstanden werden (ein Klasse steht hierarchisch über einer Methode, etc.). Weiterhin können im hier vorgestellten Ansatz Hierarchiebeziehungen zwischen Einzelmustern bei Bedarf explizit über eine Hierarchiekante modelliert werden. Im Rahmen dieser Arbeit wird angenommen, dass eine weitergehende Berücksichtigung von Hierarchieebenen die Mustererstellung unnötig erschweren würde. Der Ansatz der verteilten Graphtransformation wird nicht angewendet, da die explizite übergeordnete Spezifikation eines Netzwerkgraphen für die in der JPL zu erstellenden Muster nicht notwendig ist. Im Folgenden wird die Technik der lokalen attributierten Graphtransformation verwendet. Zur Definition dieser Transformationstechnik siehe Kapitel 5.2.

2.6. Erweiterung der bestehenden Arbeiten durch die JPL

In diesem Abschnitt wird die JPL von den zuvor beschriebenen Mustererkennungstechniken abgegrenzt. Es wird erläutert, inwieweit die kritischen Punkte der bestehenden Arbeiten in der JPL aufgenommen und gelöst werden. Auf dieser Basis wird der Beitrag der JPL im Rahmen der Domäne *Mustersuche über Quelltext* dargestellt.

Das Resultat einer Mustersuche, welche sich auf die Suche nach der Implementierung einer Anforderung im Quellcode bezieht, besitzt vier grundlegende Ausprägungen:

1. True Positive: Das Muster wurde gefunden, und die Anforderung in der Software implementiert (gewünschtes Ergebnis).
2. False Positive: Das Muster wurde erkannt, aber die Anforderung im Quelltext nicht implementiert (Fehler in der Mustersuche).
3. True Negative: Das Muster wurde nicht erkannt und auch nicht implementiert (gewünschtes Ergebnis).
4. False Negative: Das Muster wurde nicht gefunden, aber die Anforderung wurde implementiert (Fehler in der Mustersuche).

Das Spannungsfeld, in dem sich die in den vorherigen Kapiteln dargestellten Mustererkennungstechniken bewegen, liegt somit in der Balance zwischen der möglichst konkreten Spezifikation des zu suchenden Musters, um die Problematik in Punkt 2 auszuschließen, und einer abstrakten Beschreibung des zu suchenden Musters, um das in Punkt 4 beschrieben Ergebnis zu vermeiden. Weiterhin wird angestrebt im Rahmen der Musterspezifikation den Aufwand zu minimieren, d.h. Abstraktionen für die Identifikation von Implementierungsvarianten zu verwenden, ohne eine große Anzahl an false positives zu erzeugen. Auf dieser Basis werden im Folgenden die gezeigten Techniken analysiert und der Mehrwert der JPL aufgezeigt.

Zusammengefasste Kritikpunkte:

- Beschreibungen des Musters über eine XML Struktur: Die Mustererkennung in [TBR06] verwendet zur Musterbeschreibung eine XML Struktur. Die Reihenfolge in der Anweisungen implementiert werden müssen, ist hierdurch fest vorgegeben. So entsteht ein erhöhtes Risiko zur Identifikation von *false negatives*, da die Anweisungsreihenfolgen in verschiedenen korrekten Implementierungsvarianten voneinander abweichen können. Eine Erfassung der verschiedenen Varianten über einzelne Muster ist auf Grund der Variantenvielfalt zu aufwändig.
- Ausschließliche Beschreibung des Musters über den AST: Verschiedene Mustererkennungstechniken [BMD+00] setzen ausschließlich auf dem abstrakten Syntaxbaum des Quellcodes auf. Dies bedeutet, dass entweder die unterschiedlichen Implementierungsvarianten möglichst vollständig spezifiziert werden müssen, oder dass eine Vielzahl von Elementen bewusst ausgelassen werden, um den Aufwand der Mustererstellung zu begrenzen, ohne Implementierungsvarianten zu vernachlässigen. Hierdurch entsteht die Gefahr, dass die Muster zu ungenau sind und *false positives* begünstigt werden.

- Ausschließliche Beschreibung des Musters über den SDG: Weitere Ansätze [KH01], [LCH+06] nutzen zur Musterbeschreibung den Systemabhängigkeitsgraph, welcher allerdings zur Erkennung detaillierter Muster nur eingeschränkt verwendet werden kann, da er von Details der Anweisungen vollständig abstrahiert und somit *false positives* begünstigt.

- Zu abstrakte Musterdefinition: Die hier betrachteten Arbeiten, welche nicht auf dem AST oder SDG als Coderepräsentation aufbauen, basieren entweder auf Metriken [MA10], UML-Darstellungen [BBC08] oder Einzellösungen wie Fuzzy-Logik basierten Algorithmen [NWW01]. Die ausschließliche Nutzung dieser Techniken führt im Rahmen der Erkennung detaillierter Muster auf Grund ihres hohen Abstraktionsgrades mit großer Wahrscheinlichkeit zu einer erheblichen Anzahl an *false positives*.
  Ein alternativer Ansatz wird von [NSW+02] vorgeschlagen, welcher einen Aufbau von abstrakten Mustern zur Suche nach Design Pattern über an die UML-angelehnte Elemente beschreibt und detailliertere Einzelmuster in Subpattern kapselt. Implementierungsvarianten werden sowohl über einen Subpatternkatalog, als auch über optionale Bedingungen (optionale Attribute der spezifizierten Elemente) erfasst. Über die Anzahl und Wertigkeit der erfüllten Bedingungen wird nachfolgend ein Rating zur Angabe der Wahrscheinlichkeit erstellt, dass das Design Pattern korrekt erkannt wurde. Der Ansatz ist stark auf die über UML zu beschreibenden Elemente fokussiert und nicht auf die Detailebene ausgerichtet, wenngleich auch diese über einen ASG (Abstrakter Semantischer Graph, ein normalisierter AST) erfasst werden kann. Somit werden Beziehungen, die über an die UML-angelehnte Elemente nicht dargestellt werden, wie z.B. unterschiedliche Arten der Variablenübergabe, oder Abhängigkeitsbeziehungen, nicht über abstrahierende Elemente erkannt.

- Keine Möglichkeit der Musterunterteilung: Die meisten betrachteten Arbeiten zur Suche nach Mustern im Quellcode beschreiben Suchmuster lediglich auf einer Ebene ohne diese weiter zu unterteilen, so dass umfangreiche Muster schnell unübersichtlich werden. Der Mustererstellungsprozess in [NSW+02] bietet zwar eine hierarchische Strukturierung über Untermuster, verzichtet allerdings auf die Technik der sektionsbasierten Modularisierung und die Möglichkeit Beziehungen zwischen den Untermustern über Schnittstellen zu spezifizieren.

- Kaum Unterstützung des Musterspezifikationsprozesses: Die hier betrachteten Arbeiten enthalten kaum Prozessbeschreibungen, wie in einem konkreten Kontext ein Muster abgeleitet werden sollte. Es wird lediglich allgemein die Technik der Musterdefinition dargestellt.

Abgeleitet aus den oben beschriebenen Kritikpunkten werden in dieser Arbeit die folgenden Forschungsfragen betrachtet:

- Wie kann ein Suchmuster über Quellcode strukturiert erstellt und übersichtlich dargestellt werden, das eine Vielzahl an Elementen und Beziehungen enthält?

- Wie kann die Vielfalt möglicher Implementierungsvarianten zu einer Problemstellung in einem Suchmuster erfasst werden, ohne jede Variante detailliert beschreiben zu müssen, oder auf Grund einer zu hohen Abstraktionsebene *false positives* zu generieren?

Hierzu bietet die JPL folgende Lösungen:

- Die JPL bietet die Möglichkeit einer modularen Strukturierung eines komplexen Musters, wobei die einzelnen Module über Schnittstellen miteinander in Beziehung gesetzt werden. Somit können größere Suchmuster aus mehreren Modulen zusammengesetzt werden, und es ist während der Musterspezifikation nicht notwendig, durchgängig alle Elemente des großen Gesamtmusters zu betrachten.
- Zur Erstellung von Mustern in der JPL werden verschiedene Vorgehensweisen beschrieben, die im Kontext der Analyseunterstützung von Programmierübungen zur strukturierten Erstellung der Suchmuster eingesetzt werden können. Erläutert werden sowohl Top-Down- als auch Bottom-Up-Ansätze, zu Details s. Kapitel 6.
- Die JPL unterstützt die gemeinsame Nutzung von auf dem AST und dem SDG basierenden Syntaxelementen zur Spezifikation eines Suchmusters. Der Vorteil dieser Kombination ergibt sich aus der Möglichkeit Abhängigkeitsbeziehungen zwischen Anweisungen initial allgemein formulieren zu können um verschiedene konkrete Implementierungsvarianten dieser Struktur zu erfassen. Zur Vermeidung von *false positives* können diese Muster nachfolgend um JCG-Strukturen ergänzt werden.
- Durch die Einführung der JPL-Schnittstellenstrukturen wird der Aufwand der Mustererstellung reduziert, da diese Implementierungsvarianten bezogen auf die Variablenübergabe zwischen Gültigkeitsbereichen kapseln.

2.7. Zusammenfassung

In Kapitel 2 wurde der Mustererkennungsansatz dieser Arbeit von bereits bestehenden Ansätzen domänenübergreifend abgegrenzt und darauf aufbauend der Forschungsbeitrag im Umfeld der Spezifikationssprachen zur Mustererkennung im Quellcode dargestellt. Es wurde sowohl die direkte graphbasierte Mustersuche als auch ergänzend die Mustersuche und Quellcodebewertung über Metriken betrachtet. Dies sind die beiden grundlegenden Techniken welche in diesem Bereich eingesetzt werden, wobei auch auf die Kombination dieser Ansätze eingegangen wurde. Abschließend wurden verschiedene Kritikpunkte an den bestehenden Ansätze zusammengestellt und gezeigt, wie diese in der vorliegenden Arbeit behandelt werden.

Die Lösung, welche in dieser Arbeit dargestellt wird, liegt darin, dem Nutzer der Sprache JPL die Möglichkeit zu geben, auf dem AST und SDG basierende Elemente zusammen in Suchmustern verwenden zu können. Somit können, bezogen auf die Qualität der Mustererkennung und den Spezifikationsaufwand, ausgewogenere Muster erstellen werden, als dies mit nur einem Graphtyp möglich wäre. Weiterhin wird die Erstellung umfangreicher Muster durch die modulare Spezifikationstechnik der JPL und der Einführung abstrakter Schnittstellenknoten, welche Implementierungsvarianten zu Variablenübergaben kapseln, unterstützt. Diese Techniken stellen gegenüber den in diesem Kapitel betrachteten Arbeiten in ihrer Kombination Neuerungen dar, welche in der Domäne der Analyse von Übungsaufgaben eingeführt werden.

3. Quelltextrepräsentation

Für die Durchführung der Mustersuche über den Quelltext wird dieser als Graphstruktur repräsentiert. Im Folgenden werden die einzelnen Elemente, d.h. die Knoten- und Kantentypen, erläutert, welche diese Struktur, bzw. der zugeordnete Graphtyp, umfasst. Da diese Typen auch in der Syntax der JPL enthalten sind, werden somit gleichzeitig deren Basiselemente beschrieben.

Die Repräsentation des textuell gegebenen Quellcodes als Graphstruktur hat in der statischen Programmanalyse folgende Vorteile:

- Es fällt dem menschlichen Betrachter leichter Beziehungen in Graphen zu erkennen als im Fließtext, solange die angezeigte Graphstruktur nicht zu komplex ist. Um eine große Anzahl an Elementen visuell darstellen zu können, werden die Techniken der Abstraktion und der Modularisierung genutzt.
- Die Spezifikation eines Musters über grafische Elemente ist intuitiver als die Spezifikation über einen Text. Hoher Komplexität muss hier, wie zuvor bereits ausgeführt, durch den Einsatz von Abstraktionen begegnet werden.
- Algorithmen zur automatisierten Codeanalyse werden häufig auf Graphstrukturen ausgeführt, da hier die impliziten Abhängigkeiten der Elemente explizit dargestellt werden und somit einfacher erfassbar sind.

Die folgende Grafik zeigt die in diesem Kapitel vorgestellten Graphstrukturen, bezogen auf ihren Abstraktionsgrad. Während der JCG dem Quelltext sehr nah ist, abstrahiert der AJSDG von dessen syntaktischen Elementen. Der Pfad und die transitive Hülle stellen Beziehungen dar, welche auf den zuvor genannten Strukturen aufsetzen und deren Beziehungsstrukturen verallgemeinern.

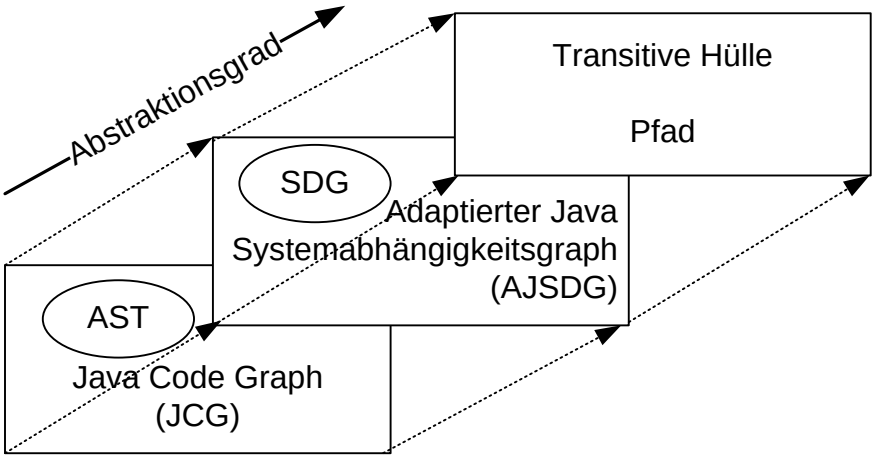


Abbildung 1: Graphtypen zur Quelltextrepräsentation

Das Kapitel beginnt mit der Repräsentation des Quellcodes als *Abstraktem Syntaxbaum*, welcher automatisch durch den Parser der Sprache Java generiert wird und somit den Startpunkt der Quelltextbetrachtung bildet. Nachfolgend wird diese Darstellung zum Java Code Graph (JCG) erweitert. Hierdurch werden u.a. Graphelemente in die Struktur eingefügt, die objektorientierte Komponenten und Beziehungen explizit erfassen, und somit die Suchmusterspezifikation auf der Quelltextrepräsentation vereinfachen.

Im Weiteren wird der Systemabhängigkeitsgraph betrachtet, welcher die Kontrollabhängigkeiten und die Datenabhängigkeiten zwischen den Anweisungen einer Applikation graphisch beschreibt, und somit vom JCG abstrahiert. Der SDG wird für den Einsatz in der JPL zum Adaptierten Java Systemabhängigkeitsgraphen (AJSDG) erweitert, um Suchmuster auf objektorientierten Strukturen spezifizieren zu können. Anschließend werden die Strukturen des Pfades und der Transitiven Hülle betrachtet, welche es ermöglichen auf JCG-Ebene und AJSDG-Ebene Beziehungen zwischen Ausdrücken zu formulieren, ohne die Elemente zwischen diesen Ausdrücken betrachten zu müssen.

3.1. Abstrakter Syntaxbaum

Der Abstrakte Syntaxbaum repräsentiert den Quelltext als Baumstruktur. Um den abstrakten Syntaxbaum zu erzeugen, wird zunächst aus einem gegebenen Quelltext ein konkreter Baum generiert, wobei die Erzeugungsregeln durch eine kontextfreie Grammatik gegeben sind. Diese legt fest, in welcher Form Elementbeziehungen des Quelltextes in der Baumstruktur repräsentiert werden. Dieser Vorgang wird von einem Parser durchgeführt, welcher im Folgenden vom konkreten Baum abstrahiert, indem Elemente gelöscht werden, deren Informationsgehalt durch die Baumstruktur bereits implizit gegeben ist. Hierbei werden sowohl alle nicht terminalen Elemente, welche während des Parsens auftreten, als auch terminale Zeichen, welche Präzedenzen von Operatoren in Ausdrücken enthalten, entfernt. Der so entstandene abstrakte Syntaxbaum ist semantisch äquivalent zum Ursprungsprogramm (Annotationsstrukturen werden hier nicht berücksichtigt). Eine Rücktransformation der abstrakten Baumstruktur zum Ursprungscode ist meist nicht mehr möglich, da vom Parser während der Abstraktion auch Elemente wie Kommentare, Leerzeichen, und explizite Klammerungen entfernt werden. Die abstrakte Syntaxrepräsentation wurde im Laufe der Zeit in verschiedenen Analysekontexten verwendet, da die Erstellung von Algorithmen zur Quelltextanalyse über diese Darstellungsart gegenüber dem Fließtext große Vorteile besitzt, wie z.B. die Möglichkeit von Variablenbenamungen zu abstrahieren (zu weiteren Details s. Kapitel 2.1).

Die in dieser Arbeit verwendete kontextfreie Grammatik basiert auf der Bibliothek des JavaCC Tools, bzw. des darauf aufbauenden JavaTreebuilder CompilerCompiler [JCC14], welcher für verschiedene Java Versionen die entsprechende Grammatik zur Erzeugung des abstrakten Syntaxbaums bereitstellt. Diese Tools werden im Weiteren für die Automatisierung des statischen Tests von Übungsaufgaben genutzt. Die Vorteile von JavaCC gegenüber anderen CompilerCompilern liegen in seiner hohen Marktdurchdringung, der kontinuierlichen Pflege der Bibliotheken durch eine starke Nutzergemeinde und seiner ausgereiften Implementierung. Zur Realisierung des in dieser Arbeit dargestellten Ansatzes wird auf die am Lehrstuhl „Spezifikation von

Softwaresystemen“ der Universität Duisburg-Essen etablierten Frameworks und Tools zurückgegriffen.

JavaCC ist ein Parsergenerator, welcher Java-Applikationen auf Übereinstimmungen mit einer gegebenen Grammatik (der Java-Spezifikation) hin überprüft. Zur Erzeugung des abstrakten Baumes wird nachfolgend in Kombination der Java Treebuilder eingesetzt, welcher die erkannten Elemente des zu untersuchenden Javacodes gemäß der Grammatik in den Baum einfügt. Die Syntax und Semantik des AST ergibt sich aus der entsprechend verwendeten Grammatik [JCC14] und bildet einen Teilgraph des im Folgenden betrachteten Java Code Graphen.

Die Erweiterung des AST zum Java Code Graph zielt darauf ab, den Graphen für die Spezifikation von Suchmustern über Java-Applikationen zu optimieren. Hierbei stehen Strukturen im Fokus, die Beziehungen darstellen, welche über die Baumdarstellung der kontextfreien Grammatik hinausgehen.

3.2. Java Code Graph

Die Elemente des AST werden in der JPL genutzt, um die Detailebene des Quellcodes als gerichtete Graphstruktur abzubilden. Der Abstrakte Syntaxbaum wird in diesem Kapitel systematisch um Elemente ergänzt, welche die Spezifikation von Suchmustern über diesen Graphen durch die explizite Darstellung von impliziten Abhängigkeitsbeziehungen vereinfachen. Der Vorteil dieser Erweiterung im Rahmen der Mustererstellung ergibt sich aus der Feststellung, dass die Spezifikation von Strukturen, welche explizit in der zu untersuchenden Struktur (dem Graphen des Java-Codes) enthalten sind, einfacher ist, als die Spezifikation von impliziten Abhängigkeiten, welche erst indirekt ermittelt werden müssen. Der JCG entspricht vorwiegend der in [Wie04] entwickelten Graphspezifikation.

Die Erweiterung des AST zum Java Code Graph verfolgt drei Ziele:

- Die Musterspezifikation wird durch eine möglichst umfangreiche Darstellung der Detailinformationen des Java-Codes, welche sowohl die einzelnen Elemente des Quellcodes, als auch besonders die Beziehungen zwischen diesen Elementen umfasst, unterstützt. Implizite Beziehungen werden explizit als Struktur dargestellt.
- Um die Musterstellung zu vereinfachen, wird größtmögliche Einfachheit im JCG angestrebt. Redundanzen werden vermieden und die Erstellung eines möglichst schmalen Elementinventars in der Syntax des JCG wird angestrebt.
- Es wird eine möglichst abstrakte Beschreibung der Syntax angestrebt, um den Anpassungsaufwand für eine Adaption des JCG an eine weitere objektorientierte Sprache zu verringern.

Als Ergebnis wurde ein Inventar von 60 Knoten- und 50 Kantentypen erstellt, durch die ein Java Programm im JCG repräsentiert wird. Nähere Erläuterungen zum größten Teil dieser Elemente sowie ihrem Bezug zur Applikationssemantik sind in [Wie04] beschrieben. Die Syntax des JCG ist im Anhang aufgeführt.

Definition des JCG: Signatur für einen attributierten JCG-Typ-Graphen

Die Signatur $\sum_{JCG} = (S, F)$ sei wie folgt definiert:

1. Sorten
    a. $S = Kn \cup Ka$
    b. Kn= {project (Unterteilung in Packages), file, class, anonymousClass, accessToAnArrayElement, referenceArrayCast, primitiveArrayCast, explicitArrayInitialisation, arrayReferenceTypeInitialization, constructorChaining, switch, break, finally, statement, continue, assignment, increment, decrement, primitiveDeclaration, objectDeclaration, primitiveArrayDeclaration, objectArrayDeclaration, cast, catch, constructor, ifthen, if, elseif, else, include, interface, for, while, do-while, methodCall, method, objectInitialization, package, return, chaining, instanceof, *Operatoren..*, synchronized, this, super, throwexception, try, accessToPrimitiveTypeVaribable, accessToReferenceTypeVariable, accessToArray, accessToStaticReference, constant, literal, false, true, comment, begin, end, BlackBox (importierte Klassen, deren Quellcode nicht verfügbar ist)}
    c. Ka = {file, package, class, include, interface, implements, extends, classMethod, memberVariable, claccConstructor, nestedClass, comment, index, indexAssignment, arrayDeclaration, dimensionLength, assignment, block, break, continue, leftOperator, rightOperator, objectDeclaration, definition, parameterDeclaration, exception, constructorChaining, body, Expression, packageImport, if, else, interfaceMethod, initialization, termination, parameter, methodDefinition, signOver, objectInitialization, constructorInvocation, subdirectory, contains, reference, declaration, instance, classReference, catch, finally, variableDeclaration, consecutive}
    d. Knoten und Kanten in attributierten Graphgrammatiken können verschiedene Attribute zugeordnet werden. In der hier verwendeten Grammatik werden Attributtypen aus der Syntax der Sprache Java eingesetzt, so können z.B. die Typen String, int oder double spezifiziert werden. Die Attribute der einzelnen Knoten- und Kantentypen sind im Anhang aufgeführt.
2. Verbindungen zwischen Knotentypen über Kantentypen

    Der JCG ist ein gerichteter Graph, d.h. er enthält ausschließlich Kanten, die von einem Quellknoten auf einen Zielknoten gerichtet sind. Im JCG sind für jeden Knotentyp die zulässigen Typen der ausgehenden und eingehenden Kanten festgelegt. Die detailierte Beschreibung erfolgt im Anhang. Als Beispiel wird die Zuordnung eines Parameters zu einem Methodenkopf betrachtet:

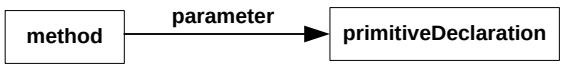


Abbildung 2: Zuordnung eines Parameters zu einem Methodenkopf im JCG

Der Knotentyp *method* repräsentiert den Kopf einer Methode. Der Parameter, welcher einen primitven Datentyp enthält, wird über einen Knoten vom Typ *primitiveDeclaration* dargestellt. Der Knoten vom Typ *method* (Quellknoten) wird über die Kante vom Typ *parameter* (Kantentyp) mit dem Zielknoten vom Typ *primitiveDeclaration* verbunden.

Grundlagen des Java Code Graphen

Alle syntaktischen Elemente werden als Knoten oder Kanten modelliert. Eigenschaften, welche die Elemente näher bestimmen, werden als Attribute der jeweiligen Elemente abgelegt. Beispielhaft kann hier der Gültigkeitsbereich (Ausprägungen: private, protected, public) oder der Name einer Variablen genannt werden. Eigenschaften sind immer genau einem Syntaxelement zugeordnet, so dass bei Änderungen das jeweilige Element isoliert betrachtet werden kann, ohne Elemente, welche mit diesem in Beziehung stehen, mit einbeziehen zu müssen.

Neben Syntaxelementen der Sprache Java, wie z.B. Methodenköpfen, Variablenaufrufen und Schleifenköpfen, werden im JCG auch die geschweiften Klammern „*{,}*“ als explizite Elemente dargestellt (*begin*- und *end*-Knoten). Weiterhin werden alle Elemente eines Ausdrucks explizit beschrieben, so dass es möglich ist, Muster für Teilausdrücke zu definieren. Im Folgenden werden die wichtigsten Zusatzstrukturen der JCG-Grammatik aufgeführt, welche über die Syntaxelemente des ASTs hinausgehen.

Zusatzelemente des JCG

- Importierte Klassen und Methoden werden als BlackBox-Element beschrieben. Diese können somit in einem Muster spezifiziert und gesucht, allerdings nicht tiefer inspiziert werden, so dass die Identifikation dieser Elemente über den entsprechenden Namen erfolgen muss und sich nicht auf innere strukturelle Eigenschaften der jeweiligen Klasse/Methode beziehen kann.
- Bei jeder Verwendung einer Variablen wird eine Beziehung vom referenzierenden Knoten zu ihrem Deklarationsknoten hergestellt. Hierbei kann es sich sowohl um eine lokale Deklaration im Methodenrumpf, eine Deklaration als Membervariable einer Klasse, oder auch um die Deklaration eines Parameters im Methodenkopf handeln. Über diese Kante ist es u.a. möglich zu bestimmen, auf welchen Gültigkeitsbereich sich eine Variable bezieht. Diese Möglichkeit wird u.a. in Kapitel 5 genutzt, um Schnittstellenvariablen zu identifizieren.
- Gültigkeitsbereiche, welche durch die geschweiften Klammern identifiziert werden, werden im Graph mit einem *Begin*-Knoten und einem *End*-Knoten gekennzeichnet. Diese Knoten werden mit einer Kante verbunden, so dass sich alle Anweisungen, die sich zwischen diesen Knoten befinden, durch Graphregeln als solche identifizieren lassen.
- Die Reihenfolge der Anweisungen wird über *Consekutiv*-Kanten repräsentiert.
- Sowohl die Parameter einer Methode als auch die Parameter eines Methodenaufrufs werden einzeln angegeben. Hierbei ist sichergestellt, dass die

ursprüngliche Reihenfolge abgebildet wird. Über diese Reihenfolge wir die spätere Zuordnung von Übergabeparametern zu Methodenparametern hergestellt.

- *If…elseif...else* Konstrukte werden mittels eines Hilfsknotens aufgeteilt und die nachfolgenden Ausdrücke entsprechend angehängt.
- Der *return*-Knoten wird mit dem Aufruf der zugehörigen Methode verbunden.

Folgendes Beispiel zeigt einen JCG-Graphen mit zugehörigem Quellcode:

class Student{
String name;
Student next;
Student (String name){
this.name=name;}}

class Studentlist{
Student head;
void insert (String s){
Student st=new Student(s);
if (head==null){
head=st;}}
}

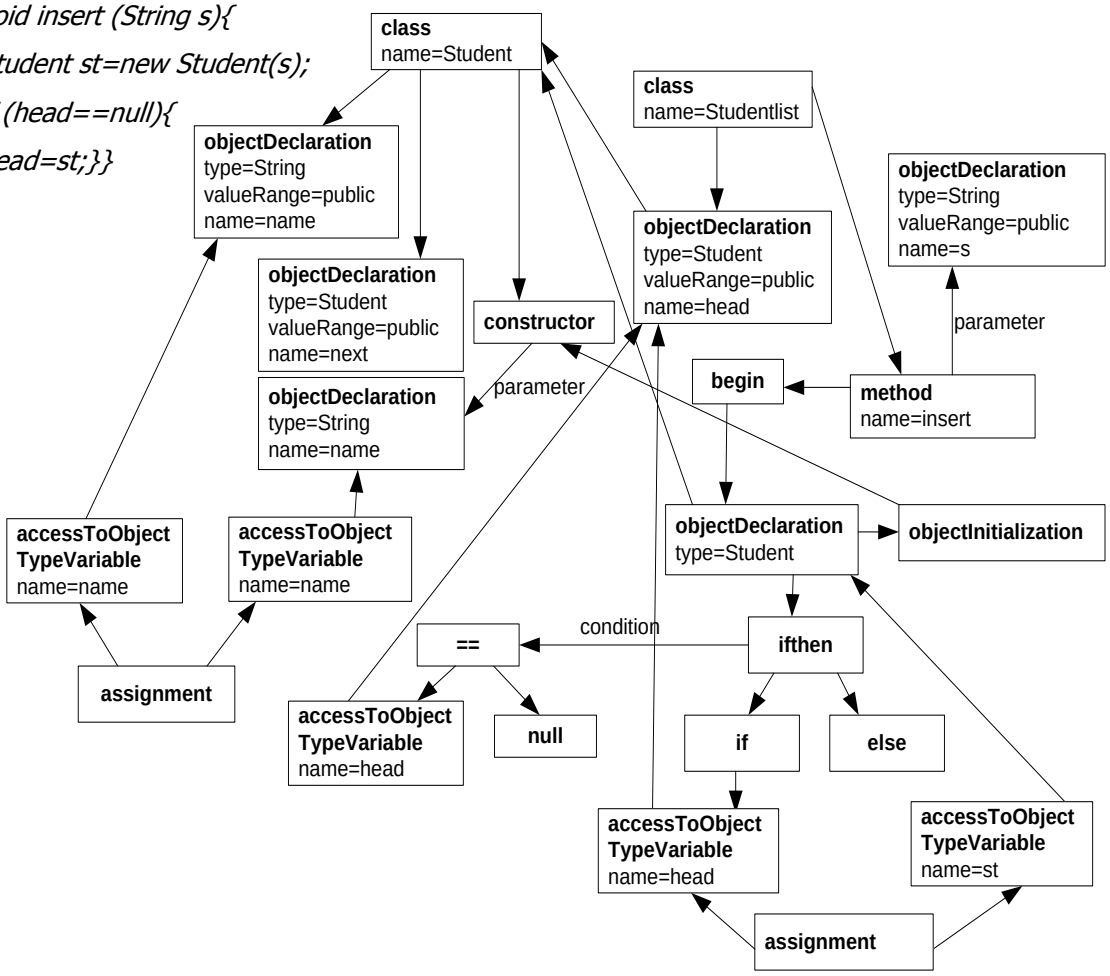


Abbildung 3: JCG-Graph zu den Klassen *Student* und *Studentlist*

Abbildung 3 zeigt den vereinfachten JCG-Quelltextgraph der Klassen Student und Studentlist. Verschiedene Elemente der vollständigen Graphstruktur wurden nicht dargestellt, um eine bessere Übersichtlichkeit zu gewährleisten, so wurden z.B. nicht alle begin- und end-Knoten aufgeführt. Die Knoten beinhalten sowohl die Angabe des Knotentyps, welcher sich auf den Typ des hierdurch repräsentierten Quelltextelements bezieht, als auch Attribute, über die seine Eigenschaften angegeben werden.

Besonders zu beachten sind die Kanten, welche von den einzelnen Variablenreferenzen auf ihre Deklaration zeigen und somit eine einfache Zuordnung einer Variablenverwendung zu ihrer Deklaration ermöglichen. Die Deklarationsknoten sind mit den Klassen des jeweils deklarierten Typs verbunden. Durch diese Kanten wird die ursprüngliche Baumstruktur der Quelltextrepräsentation im AST zu einer Graphstruktur erweitert. Die Unterteilung des Ausdrucks *Student st=new Student(s);* in *Deklarations-* und *Initialisierungsknoten* ermöglicht es zu erkennen, ob die Deklaration und die Instanziierung auf die gleiche Klasse zeigen oder ob z.B. über ein Interface deklariert wurde. Der Parameter wird hier nicht weiter betrachtet.

Die vollständige Syntax des JCG ist im Anhang aufgeführt. Der über verschiedene Tools automatisierte Prozess der Erstellung des JCG wird in Kapitel 7 erläutert.

Im folgenden Abschnitt wird die Darstellung von Anweisungsabhängigkeiten über Graphstrukturen beschrieben, so dass eine zusätzliche Sichtweise auf die Beziehungen der Quelltextelemente ermöglicht wird.

3.3. Adapted Java System Dependency Graph

In diesem Kapitel wird die Graphdarstellung der Datenabhängigkeit und der Kontrollabhängigkeit einzelner Anweisungen in einem gegebenen Quelltext betrachtet. Die Repräsentation erfolgt im Adapted Java System Dependency Graph (AJSDG), welcher die Anweisungsabhängigkeiten, die sich aus der Daten- und Kontrollabhängigkeit ergeben, sowohl auf inter- und intraprozeduraler Ebene darstellt als auch diese Beziehungen für objektorientierte Strukturen repräsentiert. Der AJSDG basiert auf dem Java System Dependency Graph, welcher in [Wal03] erläutert und in [WH07] über Graphtransformationsregeln z.T. realisiert wird. Die Struktur kann in der Quelltextrepräsentation und der hierauf aufsetzenden Suchmusterspezifikation genutzt werden, um Suchmuster auf einer höheren Abstraktionsebene als dem JCG zu beschreiben.

Vorteile der Daten- und Kontrollabhängigkeiten zur Suchmusterspezifikation (z. T. abgeleitet und erweitert aus [LCH+06]):

- Die Darstellung kann genutzt werden, um Datenabhängigkeiten und Kontrollabhängigkeiten im Suchmuster direkt erfassen zu können. Anforderungen zu Übungsaufgaben beschreiben häufig, welche Beziehungen zwischen verschiedenen Variablen implementiert werden müssen (z.B. Übergabe von Min- und Max-Parametern in einem vorgegebenen Methodenkopf, welche nachfolgend im Rahmen einer Schleife für einen Arraydurchlauf genutzt werden müssen). Somit können Suchmusterteile direkt aus diesen Anforderungen abgeleitet werden.

- Ein Vorteil dieses, verglichen mit der JCG-Darstellung, höheren Abstraktionsgrades, liegt in der Möglichkeit, mehrere Implementierungsvarianten über diese Beziehungstypen in einem Suchmuster erfassen zu können. Dies bezieht sich sowohl auf intraprozedurale Variablenabhängigkeiten als auch auf Variablenabhängigkeiten über Methoden- und Klassengrenzen hinweg.

Folgende Eigenschaften unterstützen die implementierungsvariantenunabhängige Spezifikation:

- Formatunabhängigkeit: Der AJSDG ist unabhängig von Leerzeichen und Zeilenumbrüchen.
- Variablenumbenennung: Der AJSDG ist unabhängig von Variablennamen, da lediglich Beziehungen zwischen Variablen, aber nicht die Variablen selbst betrachtet werden.
- Anweisungsreihenfolge: Die Reihenfolge von Anweisungen kann in eingeschränktem Umfang geändert werden, ohne die Darstellung des AJSDG zu verändern.
- Reihenfolgen innerhalb von Anweisungen: Die Reihenfolge von Teilausdrücken wird im AJSDG nicht betrachtet. Daraus folgt, dass Fehler auf dieser Ebene strukturell nicht erkannt werden können.
- Ersetzung von Kontrollstatements: Kontrollausdrücke können ausgetauscht werden, ohne dass dies Einfluss auf den AJSDG hat, so kann z.B. eine *while-Schleife* durch eine *for-Schleife* ersetzt werden, oder ein *if* durch ein *switch*.

Herleitung des AJSDG:

Der Programmabhängigkeitsgraph (PDG) [FOW87] beschreibt visuell Daten- und Kontrollabhängigkeiten zwischen Anweisungen einer Methode. Anweisungen werden hier als Knoten repräsentiert und die entsprechenden Abhängigkeiten als Kanten zwischen diesen Knoten abgebildet. Der Systemabhängigkeitsgraph (SDG) [HR92] erweitert den PDG um die intraprozedurale Darstellung der Daten- und Kontrollabhängigkeiten. Das Hauptanwendungsgebiet des SDG liegt im Compilerbau, wo er zur Analyse und Optimierung von Programmen eingesetzt wird.

Die Konzeption des SDG erfolgte unabhängig von speziellen Programmiersprachen, so dass es notwendig ist, diesen um programmiersprachenspezifische Strukturen zu erweitern. Die für diese Arbeit verwendete Adaption erfolgte in [Zh98, WWR03, Wal03], welche den SDG u.a. um die Repräsentation von Vererbung, Polymorphismus und Konstruktoren ergänzen. Auf dieser Struktur baut der in [WH07] konzeptionierte und über Graphtransformationsregeln realisierte Class Dependence Graph (CDG) auf, welcher im Rahmen dieser Arbeit für die explizite Darstellung von Abhängigkeiten in objektorientierten Strukturen zum AJSDG erweitert wird. Die Erweiterung wurde vorgenommen, um die Spezifikation objektorientierter Strukturen in Suchmustern weiter zu vereinfachen.

Definition der Datenabhängigkeit

Ein Knoten b ist von Knoten a datenabhängig, genau dann, wenn

1) eine Variable v existiert, welche durch die Anweisung des Knotens a belegt und in Knoten b genutzt wird. Die Anweisungen, welche sich auf dem Kontrollfluss zwischen a und b befinden, enthalten keine Zuweisung auf v.

2) eine Variable v existiert, welche durch die Anweisung des Knotens a belegt wird und zuvor in Knoten b deklariert wurde.

3) ein Objekt o existiert, welches durch die Anweisung des Knotens a instanziiert wird und zuvor in Knoten b deklariert wurde.

Definition der Kontrollabhängigkeit

Ein Knoten b ist von Knoten a kontrollabhängig, genau dann, wenn die Ausführung der in Knoten b gekapselten Anweisung abhängig ist von der Bedingung in Knoten a.

Definition des AJSDG abgeleitet aus [WWR03] und seiner Konkretisierung in [WH07]:

Die Signatur $\sum_{AJSDG} = (S, F)$ sei wie folgt definiert:

Sorten: $S = Kn \cup Ka$

- Kn ist eine Menge von Knotentypen, welche einzelne Ausdrücke der Programmiersprache Java enthalten. Kn= { *Anweisung: Eine beliebige Anweisung einer Java Applikation. Parameter: Parameter der an eine Methode übergeben wird.}*

- $Ka \subseteq Kn \times Kn$ ist eine Menge von Kantentypen, welche die Datenabhängigkeit und Kontrollabhängigkeit zwischen zwei Anweisungen repräsentieren. Ka={ *Kontrollabhängigkeit, Datenabhängigkeit, Methodenaufruf, Parameterkante, Realisierungskante* und *Vererbung*}

Verbindungen zwischen Knotentypen über Kantentypen

Der AJSDG ist ein gerichteter Graph, d.h. er enthält ausschließlich gerichtete Kanten, die von einem Quellknoten zu einem Zielknoten gehen. Im Folgenden werden sowohl jedem Kantentyp die zugehörigen Quell- und Zielknotentypen zugeordnet, als auch die Kantenrichtung spezifiziert.

- Die Kante vom Typ **Kontrollabhängigkeit** verbindet zwei Knoten vom Typ Anweisung, welche kontrollabhängig sind. Die Kantenrichtung weist von dem beeinflussenden zu dem abhängigen Knoten, d.h. in Richtung des Kontrollflusses. Die Kante beinhaltet ein Attribut *position*, welches die Position der Anweisung des abhängigen Knotens bezogen auf die Sequenz der Anweisungen im Quelltext erfasst.

- Die Kante vom Typ **Datenabhängigkeit** verbindet zwei Knoten vom Typ Anweisung und Parameter, die datenabhängig sind. Die Kantenrichtung weist von dem beeinflussenden zu dem abhängigen Knoten, d.h. in Richtung des Datenflusses.

- Die Kante vom Typ **Methodenaufruf** verbindet den Anweisungsknoten eines Methodenaufrufs mit dem Anweisungsknoten der Methodendeklaration. Die Kantenrichtung weist vom Methodenaufruf zur Methodendeklaration. Diese Kante wird auch für den Aufruf von Konstruktoren verwendet.

- Die Kante vom Typ **Parameter** verbindet die Parameterknoten eines Methodenaufrufs mit dem Anweisungsknoten des Aufrufs und die Parameterknoten der Methodendeklaration mit dem Anweisungsknoten der Methodendeklaration. Die Kantenrichtung weist vom Methodenaufruf oder der Methodendeklaration zum Parameter. Diese Kante wird auch für Parameter von Konstruktoren verwendet.

- Die Kante vom Typ **Vererbung** verbindet die Anweisungsknoten zweier Klassendeklarationen, deren Klassen in einer Vererbungsbeziehung stehen in Richtung der erbenden Klasse.

- Die Kante vom Typ **Realisierung** verbindet den Anweisungsknoten einer Interfacedeklaration mit dem Anweisungsknoten der realisierenden Klasse in Richtung der realisierenden Klasse.

Die Richtung der Abhängigkeitskanten ist gegeben durch den entsprechenden Kontroll- und Datenabhängigkeitsfluss. Es werden sowohl Abhängigkeiten innerhalb von Methoden, zwischen Methoden einer Klasse als auch zwischen Methoden verschiedener Klassen betrachtet.

Detaillierte Informationen zu dem Graphtyp CDG, auf dem der AJSDG zum größten Teil basiert, können [WH07] entnommen werden. Im Folgenden werden einige Aspekte aufgezählt, welche den AJSDG von den allgemeinen SDG-Strukturen unterscheiden, so dass dieser besser zur Spezifikation von Suchmustern eingesetzt werden kann.

- Methodenaufrufe werden unabhängig vom Gültigkeitsbereich, in den die Methode eingebettet ist, dargestellt, so dass es für die Struktur des AJSDG unerheblich ist, ob sich die aufgerufenen Methoden in der gleichen Klasse wie die aufrufende Methode oder in einer externen Klasse platziert ist. Der Vorteil dieser Darstellung ist die Möglichkeit, Methodenaufrufe in Suchmustern sehr allgemein formulieren zu können. Falls die Lokalisation einer Methode ebenfalls geprüft werden soll, so muss bei der Mustererstellung auf den zugehörigen JCG zurückgegriffen werden (sh. Abschnitt 4.1).
- Bei Vererbungsbeziehungen werden vererbte Methoden mit beiden Klassen (Ober- und Unterklasse) über eine Kontrollabhängigkeitskante verbunden, so dass die Kanten beider Klassen auf die gleiche Methode verweisen, d.h. es wird kein Duplikat für die erbende Klasse erstellt.
- Interfaces werden äquivalent zu Klassen dargestellt, d.h. auch Methoden in Interfaces werden durch entsprechende Methodenkopfknoten und mögliche

Parameterknoten repräsentiert. Die Verbindung zwischen einer Methode im Interface und der diese Schnittstelle implementierenden Methode wird durch eine *Realisierungskante* erstellt. Äquivalent werden Beziehungen zu Methoden abstrakter Klassen realisiert.

- Membervariablen werden in die Datenabhängigkeitsbeziehungen mit aufgenommen, d.h. ihr Einfluss auf und ihre Beeinflussung von Anweisungen aus der eigenen oder einer externen Klasse wird dargestellt. Die Repräsentation erfolgt über Datenabhängigkeitskanten. Die Strukturen werden beispielhaft in Abbildung 5 aufgeführt.
- Selbstreferenzen *(i++)* sind datenabhängig von der letzten Belegung der verwendeten Variablen.
- Wird eine Variable belegt, so wird eine Datenabhängigkeitsbeziehung zur Deklaration dieser Variablen generiert. Durch diese Beziehung kann in Mustern mit hierarchischen Strukturen erkannt werden, ob eine Variablenbelegung in der unteren Hierarchieebene Auswirkungen auf die obere Ebene hat. Ein Spezialfall wurde zuvor im Rahmen des Einbezuges von Membervariablen erläutert.
- Es wird die Position der kontrollabhängigen Anweisungen erfasst, um Suchmuster erstellen zu können, die sich auf Anweisungsreihenfolgen beziehen.

In der folgenden Abbildung wird ein intraprozeduraler AJSDG dargestellt, welcher zwei Methoden sowie einen Methodenaufruf enthält. Der Code realisiert einen Teil des Algorithmus zur Berechnung des „größten gemeinsamen Teilers“.

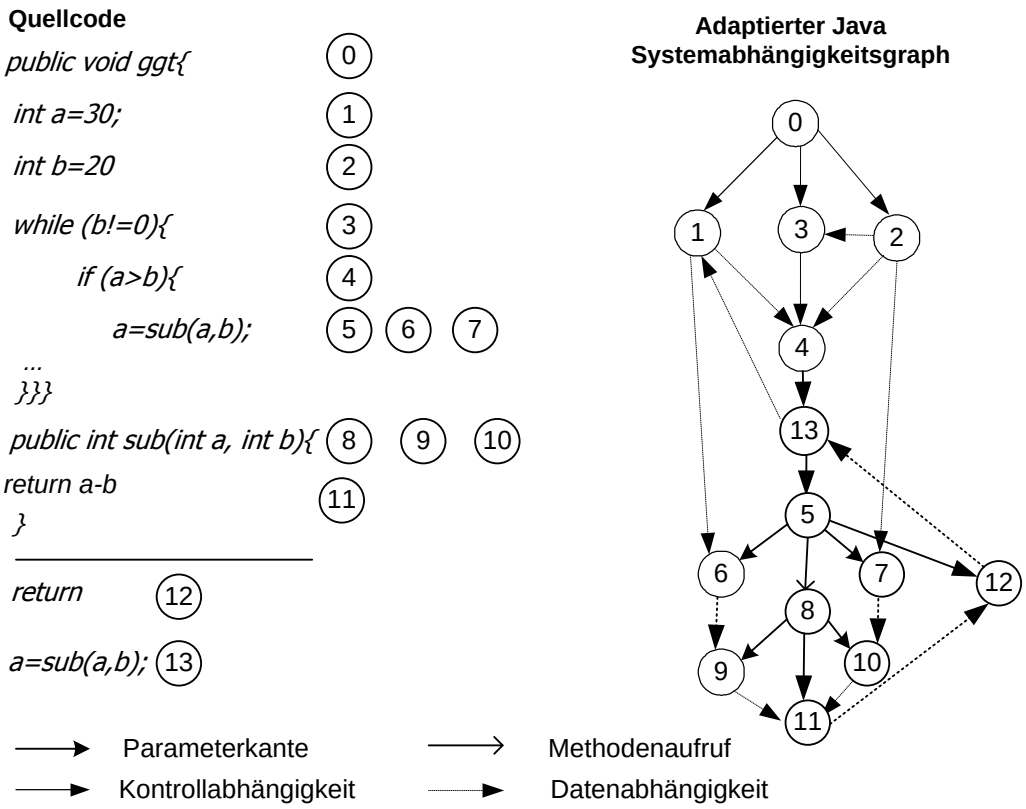


Abbildung 4: Intraprozeduraler Abstrakter Java Systemabhängigkeitsgraph

Der intraprozedurale Systemabhängigkeitsgraph in Abbildung 4 stellt die Kontroll- und Datenabhängigkeiten des Programms *ggt* visuell dar. So ist die *while*-Schleife datenabhängig von der Belegung der Variablen b. Die *if*-Anweisung ist kontrollabhängig von der vorhergehenden *while*-Schleife und datenabhängig von den Belegungen der Variablen *a* und *b*. Zu beachten ist, dass Kontrollabhängigkeiten ohne Betrachtung der Anweisungsreihenfolge im Quellcode erstellt werden. Dies bedeutet, dass alle von einem Knoten kontrollabhängigen Knoten diesem direkt zugeordnet werden, ohne dass ihre Reihenfolge im Programmablauf betrachtet wird. Dies unterscheidet den Kontrollabhängigkeitsgraph vom Kontrollflussgraphen, der diese Reihenfolge mit einbezieht.

Der Methodenaufruf wird über mehrere Knoten realisiert: Knoten 13 repräsentiert den vollständigen Ausdruck des Methodenaufrufs. Knoten 5 repräsentiert den reinen Methodenaufruf (*add*) und die zwei Knoten 6 und 7 stellen die zu übergebenden Parameter dar. Diese Elemente werden mit den entsprechenden Knoten verbunden, welche den Methodenkopf (K8) mit seinen Parametern (K9, K10) beschreiben. Die Rückgabe des Wertes aus der Methode *add* wird über die Datenabhängigkeitskante von Knoten 11 (*return*-Ausdruck innerhalb der Methode add) zu Knoten 12 (Hilfsknoten, mit dem die eingehenden Rückgabewerte verbunden werden) realisiert. Durch die Verbindung des *return*-Knotens (K12) mit dem vollständigen Ausdruck des Methodenaufrufs (K13) über eine Datenabhängigkeitskante wird die Datenabhängigkeit der Variablen a, die mit dem zurückgegebenen Wert belegt wird, dargestellt.

Im Folgenden wird die Repräsentation grundlegender objektorientierter Strukturen im AJSDG beschrieben.

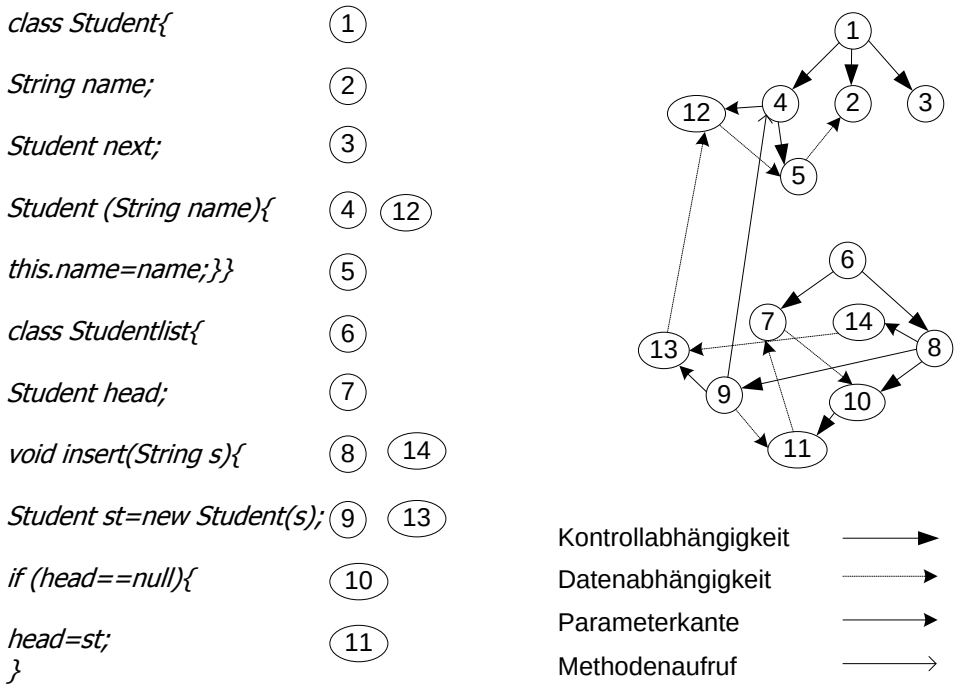


Abbildung 5: Java Systemabhängigkeitsgraph mit objektorientierten Strukturen

Im zuvor aufgeführten Beispiel wird gezeigt, wie die Beziehung zwischen einer Objektinstanziierung und dem hieraus resultierenden Konstruktoraufruf im AJSDG repräsentiert wird. Der Aufruf des Konstruktors im Rahmen der Objektinstanziierung wird analog zu einem Methodenaufruf vorgenommen, mit entsprechender Verbindung zwischen dem Aufrufausdruck (KN9) und dem Konstruktorkopfknoten (KN4) und der Parameterkante zwischen den Knoten der Parameterübergabe (KN13 nach KN12). Im AJSDG wird strukturell nicht zwischen dem Aufruf von Methoden und Konstruktoren unterschieden. Weiterhin ist an der Datenflusskante zwischen (KN11) und (KN7) zu sehen, dass die Membervariable *head* der Klasse *Studentlist* abhängig ist von der Zuweisung *head=st*.

Nachdem in diesem Kapitel der Aufbau des AJSDG und dessen Ableitung aus SDG und CDG dargestellt wurde, wird im folgenden Abschnitt die transitive Hülle betrachtet, welche über dem AJSDG aufgespannt wird.

3.4. Transitive Hülle

In diesem Kapitel wird die Struktur der transitiven Hülle vorgestellt, welche auf dem Adaptierten Java Systemabhängigkeitsgraphen aufsetzt. Durch die transitive Hülle werden alle ASDG-Knoten, welche durch eine beliebige Anzahl von Relationen miteinander in Beziehung stehen, direkt miteinander verbunden.

Die transitive Hülle wird auf dem Datenabhängigkeitsgraphen und dem Kontrollabhängigkeitsgraphen angewendet, so dass während der Suchmustererstellung alle indirekten Anweisungsabhängigkeiten direkt über „transitive Kanten“ spezifiziert werden können. Im Rahmen der Spezifikation von Datenabhängigkeiten vereinfacht dies u.a. die Abstraktion über Abhängigkeitsketten, wie sie z.B. beim Einsatz von Hilfsvariablen auftreten.

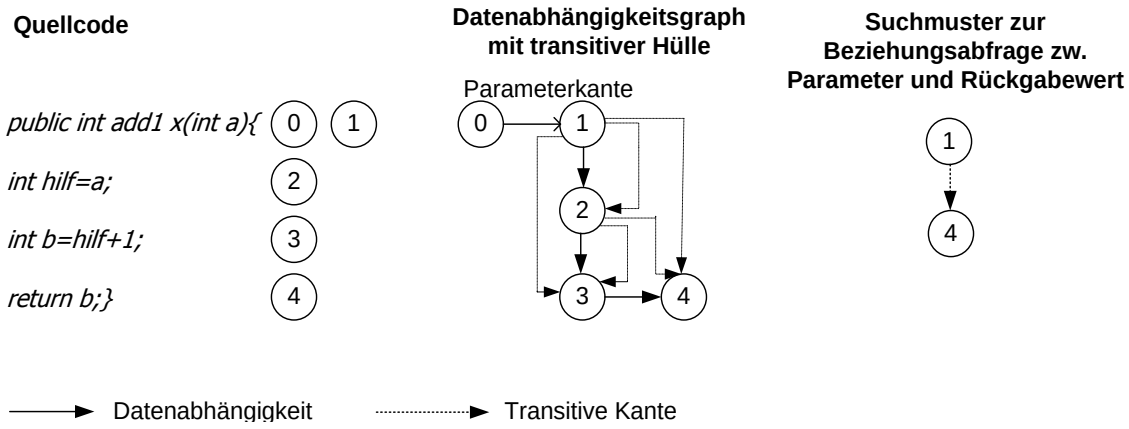


Abbildung 6: Transitive Hülle über dem Datenabhängigkeitsgraphen

In Abbildung 6 wird der Quellcode zur Erhöhung einer Variablen um 1 in der Methode add1 dargestellt. Der Wert des Parameters wird zuerst an eine Hilfsvariable übergeben und nachfolgend erhöht, bevor dieser wieder zurückgegeben wird. Rechts vom Quelltext wird der entsprechende Datenabhängigkeitsgraph zusammen mit der

transitiven Hülle dargestellt. Durch die Nutzung der Beziehungen der transitiven Hülle kann mit dem einfachen Suchmuster rechts ermittelt werden, ob der Methodenparameter den Rückgabewert der Methode beeinflusst.

Durch den Einsatz der transitiven Hülle im Kontrollabhängigkeitsgraphen kann von geschachtelten Bedingungsstrukturen abstrahiert werden:

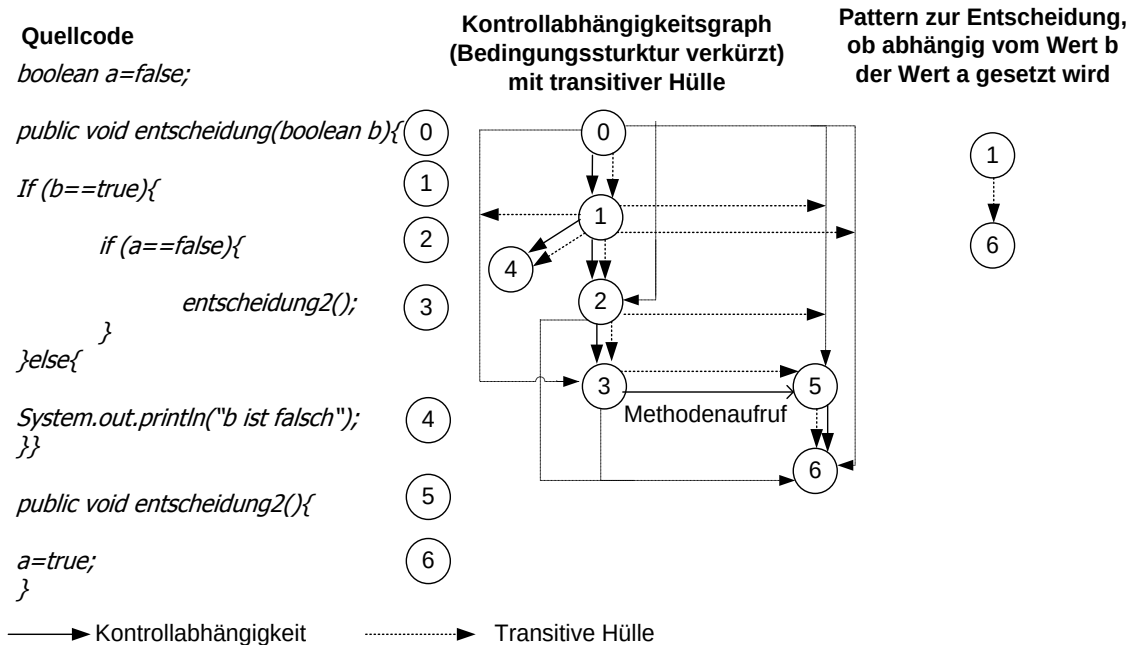


Abbildung 7: Transitive Hülle über dem Kontrollabhängigkeitsgraphen

In Abbildung 7 wird ein Quellcode gezeigt, der verschachtelte Bedingungen enthält. Rechts vom Quelltext ist der Kontrollabhängigkeitsgraph und die zugehörige transitive Hülle dargestellt. Ein Muster, das nach einer Kontrollabhängigkeit zwischen zwei Anweisungen sucht, kann durch Einbezug der transitiven Hülle Abhängigkeiten spezifizieren, ohne alle Verschachtelungsvarianten und die hiermit zusammenhängenden Bedingungen betrachten zu müssen. Somit kann bereits durch die einfache Regel rechts in der Darstellung eine Kontrollabhängigkeit zwischen *a* und *b* formuliert werden, d.h. es kann geprüft werden, ob die Belegung der Variablen *a* abhängig ist vom Wert der Variablen *b*.

Die Realisierung der transitiven Hülle über der Graphrepräsentation des Quelltextes durch Graphtransformationsregeln wird in Kapitel 5.8 dargestellt.

3.5. Pfad

Das JPL Element *Pfad* hat im Kontext des JCG eine ähnliche Funktion wie die transitive Kante im Kontext des AJSDG, da über diesen ebenfalls indirekte Beziehungen direkt spezifizierbar werden. Die Verwendung der Pfad-Struktur in einem JPL-Suchmuster ermöglicht die Spezifikation einer Reihenfolge von mehreren JCG-Elementen im Suchmuster, welche auch dann erkannt wird, wenn diese im zu untersuchenden Quelltext nicht direkt aufeinanderfolgen. Im Gegensatz zur transitiven Kante ist der Pfad auf einen Gültigkeitsbereich beschränkt, d.h. die Anfangs- und der

Endknoten, welche durch den Pfad verbunden werden, müssen sich im gleichen Gültigkeitsbereich bzw. einem in diesen eingebetteten Sub-Gültigkeitsbereich befinden, um durch die Pfad-Struktur gefunden zu werden.

Einschränkung: Die JCG-Knoten dürfen nicht in der gleichen Anweisung auftreten.

Informal Definiert: Durch den Pfad werden alle JCG-Elemente eines Gültigkeitsbereichs, welche durch eine beliebige Anzahl von Relationen miteinander in Beziehung stehen und nicht Teil der gleichen Anweisung sind, direkt miteinander verbunden.

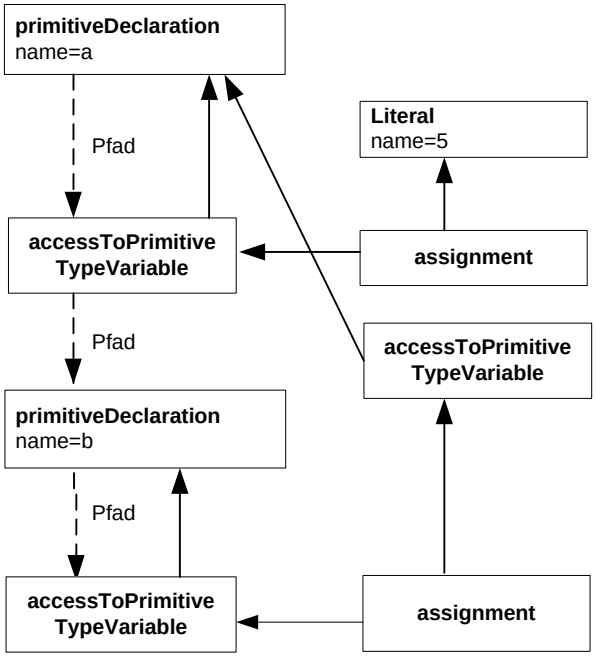


Abbildung 8: Suchmuster zur Erläuterung des Pfad-Elements in der JPL

In diesem Beispiel wird ein JPL-Suchmuster beschrieben, in dem eine Variable *a* zuerst deklariert, und danach mit dem Wert *5* belegt wird. Nachfolgend wird eine Variable *b* deklariert, und dieser der Wert der Variablen *a* zugewiesen.

Folgendes Codefragment wird durch dieses Muster erfasst:

int a;
…beliebige Anzahl von Anweisungen.
a=5;
…beliebige Anzahl von Anweisungen.
int b;
… beliebige Anzahl von Anweisungen.
b=a;

Die Realisierung des Pfades während der Ausführung der Mustersuche basiert auf der Struktur der transitiven Hülle. So werden zuerst die beiden zugehörigen AJSDG-Knoten der Elemente gesucht, welche durch den Pfad verbunden sind. Nachfolgend wird untersucht, ob eine „transitive Kontrollabhängigkeitskante“ zwischen den AJSDG-Knoten existiert. Die Realisierung der Pfadstruktur über Graphtransformationsregeln wird in Kapitel 5.9 dargestellt.

3.6. Zusammenfassung

In Kapitel 3 wurde die graphbasierte Repräsentation des Quelltextes betrachtet, auf der die Suchmuster nachfolgend aufsetzen. Es wurde sowohl die Syntax des quelltextnahen JCG als auch des AJSDG dargestellt, welcher die Kontroll- und Datenabhängigkeiten der objektorientierten Sprache Java erfasst. Die vollständige Syntax des JCG ist im Anhang aufgeführt. Weiterhin wurden die Strukturen der transitiven Hülle und des Pfades beschrieben, die es ermöglichen, indirekte Beziehungen zwischen Syntaxelementen direkt darzustellen. Da diese Elemente der Quelltextrepräsentation auch direkt in einem JPL-Suchmuster eingesetzt werden können, wurde in diesem Kapitel somit implizit bereits ein großer Teil der JPL-Sytax vorgestellt.

Umfangreichere Suchmuster, die direkt auf den hier vorgestellten Syntaxelementen beruhen, werden schnell unübersichtlich und somit anfällig für Spezifikationsfehler. Aus diesem Grund wird in der JPL ein Konzept der modularen Musterspezifikation eingeführt, welches auf der Strukturierung des Java-Quellcodes in Gültigkeitsbereiche basiert. Im nächsten Kapitel wird sowohl die modulare Struktur der JPL betrachtet, als auch die JPL-Schnittstellenstruktur eingeführt, welche die Spezifikation von Variablenübergaben zwischen diesen Bereichen vereinfacht.

4. Java Pattern Language

Die Java Pattern Language [KG08] ermöglicht die Spezifikation von Suchmustern, welche auf der im vorhergehenden Kapitel dargelegten Graphrepräsentation des Quellcodes basieren. Voraussetzung für die Mustersuche ist, dass der Quelltext syntaktisch entsprechend der Java Grammatik korrekt ist. Der Fokus der JPL liegt in der Unterstützung der Formulierung von Suchmustern, welche Codestrukturen über mehrere Gültigkeitsbereiche hinweg enthalten.

Dieses Kapitel beschreibt die Syntax und Semantik der JPL. Abbildung 9 zeigt die Integration der zuvor beschriebenen Quelltextrepräsentationen in die JPL. Diese werden ergänzt um die modulare Darstellung von Suchmustern und vom JCG abstrahierenden Schnittstellenelementen, welche nachfolgend eingeführt werden.

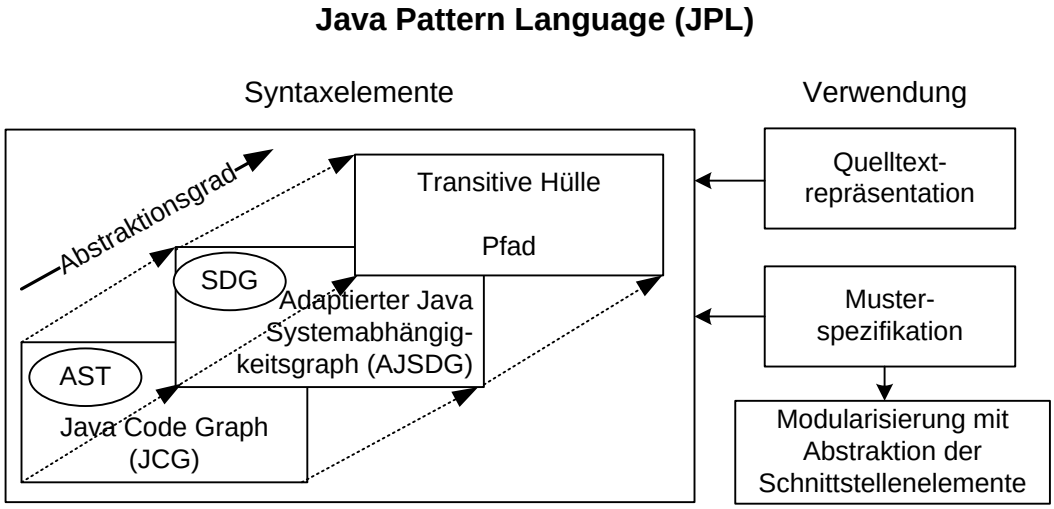


Abbildung 9: Syntaxelemente und deren Verwendungskontexte in der JPL

Das Ziel der JPL ist die Bereitstellung einer Musterspezifikationssprache, welche folgende Eigenschaften unterstützt:

- Verwendung der in Kapitel 3 eingeführten Elemente, um mit möglichst geringem Aufwand Suchmuster spezifizieren zu können, welche viele Implementierungsvarianten zu einer Anforderung enthalten.
- Erstellung komplexer Suchmuster aus einzelnen Modulen, um umfangreiche Muster mit vielen Graphelementen übersichtlich spezifizieren zu können.
- Unterstützung der Spezifikation komplexer Muster durch Einführung der JPL-Schnittstellenstruktur, welche Implementierungsvarianten (verschiedenen Arten der Variablenübergaben), die sich aus der Kopplung einzelner Module ergeben, kapselt.

Zur Darstellung komplexer Suchmuster in einem einheitlichen Modell wird das Modularitätsmodell von Goedicke und Schumann [GS94] für die Visualisierung von Suchmustern adaptiert. Durch diesen Ansatz können einzelne Muster über die Nutzung von Schnittstellensektionen zu einem komplexen Muster kombiniert werden. Die

Verbindung der Module erfolgt entweder über die zuvor erläuterten Elemente des JCG und AJSDG, oder über JPL-Schnittstellenknoten, welche in Abschnitt 4.4. eingeführt werden.

Das Kapitel beginnt mit einem Überblick über die Syntax der JPL. Nachfolgend wird gezeigt, wie die Elemente des JCG gemeinsam mit Elementen des AJSDG zur Erstellung von Suchmustern verwendet werden. Der folgende Abschnitt 4.2. diskutiert Ansätze zur modularen Softwarespezifikation, welche nachfolgend in Abschnitt 4.3. auf die Mustererkennung übertragen und in die JPL integriert werden. In Abschnitt 4.4. werden die in den Schnittstellensektionen zu verwendenden JPL-Schnittstellenknoten beschrieben. Die Kopplung selbst wird in Abschnitt 4.5. erläutert, wobei in Abschnitt 4.6. die Darstellung hierarchischer Strukturen erklärt wird. In Abschnitt 4.7. wird auf die Repräsentation semantisch zusammenhängender Module eingegangen, während die Abschnitte 4.8. und 4.9. beispielhaft die Kopplung von Modulen unter Verwendung der Elemente des JCG und AJSDG beschreiben.

Im folgenden Kapitel werden die Sprachelemente der JPL eingeführt, welche zur Musterspezifikation verwendet werden können.

4.1. Überblick über die Syntax der JPL

Die JPL beinhaltet ein großes Repertoire verschiedener Elemente, so dass der Nutzer die Möglichkeit hat, Suchmuster zu gegebenen Anforderungen in verschiedenen Ausprägungen und Detailierungstiefen zu spezifizieren. Die Beschreibungsmöglichkeiten reichen von sehr detaillierten Suchmustern unter Verwendung der JCG-Elemente und der hieraus resultierenden Notwendigkeit explizit die unterschiedlichen Implementierungsvarianten zu erstellen, bis hin zu sehr abstrakten Suchmustern und der sich hieraus ergebenden erhöhten Wahrscheinlichkeit von *false positives*. Verschiedene Methoden zu Erstellung von Suchmustern werden in Kapitel 6 beschrieben.

Die Syntax der JPL, bzw. des JPL-Graphen umfasst folgende in Kapitel 3 definierten Elementmengen:

- JCG: Elemente des Java Code Graph (s. Abschnitt 3.2)
- AJSDG: Elemente des Adapted Java System Dependency Graph (s. Abschnitt 3.3)
- KaT: die transitiven Kanten (s. Abschnitt 3.4)
- KaP: die Kante Pfad**,** (s. Abschnitt 3.5)

Nachdem die grundlegenden Elemente aufgeführt wurden, welche in einer JPL-Spezifikation verwendet werden können, wird im Folgenden der modulbasierte Aufbau des Suchmusters, bzw. des hieraus resultierenden JPL- Graphen (JPLG), über den das Muster repräsentiert wird, definiert.

Definition des JPL -Graphen:

1. Sorten:
    1. $JPLG = \{JPLM \cup MVBKa \cup \text{MVBKaSZ} \cup \text{Hierarchiekante}\}$ Der Graph ist unterteilt in Module (JPLM). Die Module werden durch Modulverbindungskanten (MVBKa, MVBKaS) sowie über den Kantentyp der Hierarchiekante miteinander verbunden.
    2. $JPLM = \{ID, I, E, K\}$: Die einzelnen Module des JPLG sind unterteilt in die Sektionen: Identifikation (ID), Import(I), Export(E), Körper(K).
    3. ID= $K = \{JCG \cup AJSDG \cup KaP \cup KaT\}$: Definition, welche Elementmengen in den Sektionen ID und K verwendet werden dürfen.
    4. $I = E = \{JCG\_SK \cup AJSDG\_SK \cup JPLSK\}$**:** Definition, welche Elementmengen in den Sektionen I und E verwendet werden dürfen.
    5. JCG_SK={objectDeclaration, primitiveDeclaration, arrayDeclaration, accessToReferenceTypeVariable, accessToPrimitiveTypeVariable, accessToArray}: Schnittstellenknoten aus der JCG-Elementemenge
    6. AJSDG_SK={Anweisungsknoten, Parameterknoten}: Schnittstellenknoten aus der AJSDG-Elementmenge
    7. JPLSK: {JPLPrimitive, JPLObject, JPLVariable, JPLSpecific}: JPL-Schnittstellenknoten zur abstrakten Definition von Datenabhängigkeitsbeziehungen zwischen Modulen
    8. MVBKaS={else, if, case, default, catch}: Modulverbindungskanten für semantisch zusammenhängende Blöcke
    9. Hierarchiekante: Kante zur Repräsentation von Hierarchiebeziehungen zwischen Modulen. Ein Modul a, dessen Gültigkeitsbereich den eines weiteren Moduls b umfasst, steht hierarchisch über diesem.
    10. SKa: Schnittstellenkante, die Knoten vom Typ JPLSK in der Importschnittstelle eines Moduls mit Knoten vom Typ JPLSK in der Exportschnittstelle eines weiteren Moduls verbindet.
    11. MVBKa = {Deklarationsverbindung, Referenzverbindung, Datenabhängigkeitskante, transitive Kante für Datenabhängigkeit, SKa} : Modulverbindungskante; Zusammenfassung der Kanten, die Knoten in der Importschnittstelle eines Moduls mit Knoten in der Exportschnittstelle eines weiteren Moduls verbinden.

2. Im Folgenden werden Funktionen zur Ableitung des JPL-Graphen in eine Menge von ausschließlich aus JCG und AJSDG-Elementen bestehenden Graphen dargestellt. Diese Graphstrukturen können nachfolgend in der Graphrepräsentation des zu untersuchenden Quelltextes gesucht werden, die ebenfalls ausschließlich durch Elemente des JCG und AJSDG dargestellt wird.

Notationserläuterung:

- a_b : Graphstruktur a, die Knoten und Kanten vom Graphtypen b enthält.
- a(b): Ein Knoten des Typs a, der in der Modulsektion b spezifiziert ist.
- a – b – c: Ein Knoten des Typs a wird über eine Kante des Typs b mit einem Knoten des Typs c verbunden.
- $f : a \rightarrow \{b\}$:f ist eine Funktion über die ein Graph vom Typ a in eine Menge von Graphen des Typs b abgeleitet wird.

Funktionen:

- IEG_{JCG}**:** Graphstruktur, welche eine Import/Export-Datenabhängigkeitsbeziehung zweier Gültigkeitsbereiche über JCG Elemente realisiert.

1. $f : (JPLSK(Export) - SKa - JPLSK(\mathrm{Im}\, port)) \rightarrow \{IEG_{JCG}\}$**:** Schnittstellenknoten JPLSK in der Modulsektion Export werden über die Schnittstellenkante mit den Schnittstellenknoten in der Importsektion des referenzierten Moduls verbunden und somit wird eine abstrakte Datenübergabebeziehung spezifiziert. Diese Import/Export- Struktur des JPL-Graphen, welche von den verschiedenen Möglichkeiten Daten über Gültigkeitsbereiche hinweg zu übergeben abstrahiert, wird in eine Menge von Import/Export-Graphstrukturen abgeleitet, welche die verschiedenen Implementierungsvarianten direkt über JCG-Elemente darstellen. Die Realisierung ist in Kapitel 5.4 beschrieben.
2. $f : JPLG \rightarrow \{JCG \cup AJSDG\}$**:** Der modularisierte JPL Graph wird in eine Menge nicht modularisierter Muster abgeleitet, welche die in diesem Graph gekapselten Implementierungsvarianten ausschließlich über Elemente der JCG und AJSDG Syntax beschreiben. Hierzu werden die abstrakten Datenübergabevarianten aufgelöst (s. vorherige Funktion), und eine Regelreihenfolge abgeleitet, welche die Suchreihenfolge für die einzelnen hieraus entstehenden Teilmuster festlegt. Hierbei müssen sowohl die Eingrenzung des Suchraums durch Muster in der Identifikatorsektion als auch Beziehungen zwischen den Schnittstellensektionen beachtet werden. Die Realisierung der Ableitung und der Algorithmus, über den die einzelnen Muster der hier erstellten Graphmenge auf einem gegebenen Quellcodegraphen gesucht werden, ist in den Kapiteln 5.9 und 5.10 beschrieben.

Die hier definierten Sorten des JPL-Graphen, sowie deren Verwendung, werden in den folgenden Unterkapiteln ausführlich erläutert. Die Realisierung der oben definierten Funktionen wird in Kapitel 5 betrachtet.

Kombinierte Spezifikation von JCG- und AJSDG-Syntaxelementen

Die Basis der JPL bilden die Elemente zur Quelltextrepräsentation, welche in Kapitel 3 dargestellt wurden. Die JPL ermöglicht es, alle Elemente gleichzeitig in einer Musterspezifikation einzusetzen. Hinzu kommt, dass einzelne Elemente des JCG und AJSDG sich in einer JPL-Spezifikation aufeinander beziehen können, um hierdurch die Erfassung verschiedener Implementierungsvarianten zu unterstützen. So können zu

starke Abstraktionen, welche z.B. in [LCH+06] unter ausschließlicher Verwendung des SDG zu einer starken Ungenauigkeit der Suchmuster geführt haben, durch die zusätzliche Verwendung von JCG-Elementen vermieden werde, ohne die Lösungsvarianten zu stark einzuschränken.

Zur Suchmusterspezifikation in der JPL kann der AJSDG gemeinsam mit dem JCG verwendet werden. Die Beziehungen, die zwischen den Elementen der beiden Graphstrukturen bestehen, werden über IDs spezifiziert, welche als Attribut in jedem Knoten enthalten sind. Durch die Verwendung von IDs wird die Übersichtlichkeit des Suchmusters über den Gesamtgraphen gewahrt, welche durch zusätzliche Elemente, wie z.B. Beziehungskanten, eingeschränkt worden wäre. Zwischen den Knoten des AJSDG und den Elementen des JCG besteht hierbei eine *1:n* Beziehung.

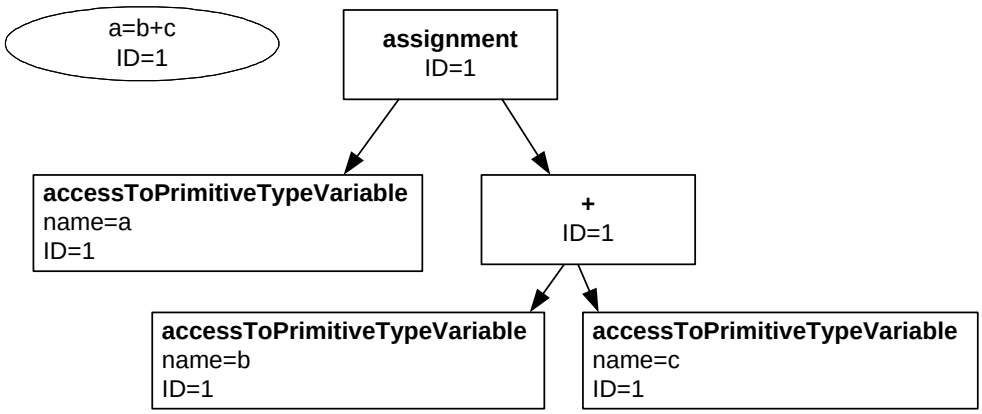


Abbildung 10: Beziehung zwischen JCG und AJSDG

In Abbildung 10 wird die Spezifikation des Ausdrucks *a=b+c* dargestellt. Die JCG-Knoten des Ausdrucks enthalten hierbei eine ID, die identisch ist mit der ID des AJSDG-Knotens.

Da zwischen JCG- und AJSDG-Elementen keine direkte strukturelle Kopplung besteht, lösen Änderungen in einem der beiden Graphen keine Anpassungen am entsprechend anderen Graphen aus. Um Konsistenzprobleme zu vermeiden, müssen daher bei Transformationen, die im Rahmen der Mustersuche evtl. notwendig sind, immer beide Graphen gemeinsam betrachtet werden.

Im Folgenden wird die modulare Struktur der JPL hergeleitet.

4.2. Überlegungen zur Modularität

Bereits die Beschreibung von Mustern geringer Komplexität, wie z.B. das Einfügen eines Elements in eine vorgegebene Liste, erfordert ein hohes Maß an Sorgfalt, um eine möglichst große Zahl an Implementierungsvarianten strukturiert erfassen zu können [KG06]. Daraus resultiert, dass für die Entwicklung komplexer Suchmuster unterstützende Strukturen erforderlich sind, um die Übersicht über die Einzelelemente sicherzustellen und Inkonsistenzen im Muster entdecken zu können. In diesem Abschnitt wird das Konzept der Modularität im Kontext der Komponentenspezifikationssprache II [GS94] vorgestellt, welches nachfolgend in der JPL zur Unterstützung der Musterspezifikation adaptiert wird.

Modularität in der Architekturspezifikationssprache II nach [GS94]:
Eine Softwarekomponente ist eine Einheit, welche sowohl Dienste anbietet, als auch Dienste anderer Komponenten nutzt. Sie kapselt die lokalen Strukturen, welche zur Realisierung des Services eingesetzt werden, so dass durch die Darstellung als Modul eine neue Abstraktionsebene eingeführt wird, welche hilft, die Komplexität von großen Softwaresystemen zu bewältigen indem diese aus mehreren einzelnen Modulen zusammengesetzt werden. Die Verbindung erfolgt über Schnittstellen, welche sowohl die geforderten Parameter festlegen als auch die entsprechenden Rückgabewerte beschreiben.

Das Konzept der Modularisierung wird in dieser Arbeit auf die Spezifikation von Suchmustern übertragen. Es werden hierfür Teile des für die Spezifikationssprache II [GS94], [GSC91] entwickelten Modularisierungskonzepts genutzt, welches im Folgenden kurz beschrieben wird. Das Ziel der Adaption dieser Technik auf die Suchmusterstruktur liegt in der konsistenten Nutzung von Interfaces für die Kopplung einzelner Suchmuster zu komplexeren Strukturen.

Die Komponentendarstellung in II

Ein Modul besteht aus 4 Sektionen:

- In der Importschnittstelle werden die Dienste beschrieben, welche von externen Modulen importiert werden.
- Im Export werden die Dienste beschrieben, welche von der Komponente nach außen angeboten werden, so dass diese von externen Komponenten genutzt werden können.
- Im Body wird die Realisierung der Dienste der Komponenten beschrieben.
- In den Common Parameters werden Datentypen aufgeführt, welche importiert und danach unverändert exportiert werden.

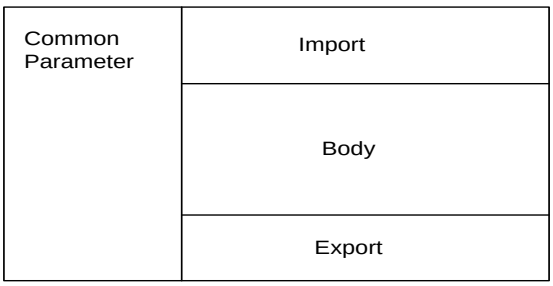


Abbildung 11: Komponentendarstellung in II

Die Kopplung der einzelnen Komponenten erfolgt über eine Konfiguration. Hier werden die im Import definierten Schnittstellen mit ihren Gegenstücken im Export der anbietenden Komponenten verbunden. Über diese Technik kann die Konsistenz der erstellten Komponentenstruktur sichergestellt werden, da mit jeder Signatur einer Importschnittstelle mindestens eine gültige Signatur im Export eines Moduls verbunden sein muss.

Der Modulansatz der JPL verbindet die Moduldarstellung in II mit der Darstellung der Suchmusterspezifikationen, d.h. des JPL-Graphen, wobei die Sektion der *Common Parameters* nicht verwendet wird. Die Adaption der Modulstruktur wird im folgenden Kapitel beschrieben.

4.3. Modulrepräsentation der JPL

Die zuvor betrachtete Strukturierung eines Moduls in unterschiedliche Sektionen wird im Weiteren auf die Darstellung von Suchmustern über Java Quellcode in der JPL übertragen.

Ein Modul der JPL bezieht sich immer auf einen Gültigkeitsbereich (Block). Ein Block definiert den Gültigkeitsbereich von Variablen, welcher in der Java Syntax durch geschweifte Klammern {} umfasst wird.

Während eine Architekturkomponente im Kontext der Programmiersprache Java meistens eine Klasse oder ein Package (Gruppe von Klassen) beschreibt, wird es durch die Umwidmung der Semantik möglich, auch detaillierte Strukturen wie Methoden, Schleifen, Bedingungen, etc. mit dieser Darstellung zu erfassen, wobei die zuvor erwähnten Blöcke der Klasse und des Packages durch die gleiche Darstellungsmethode spezifiziert werden können. Ein weiterer Vorteil der Definition eines Moduls als Block ist die einheitliche Betrachtung sowohl von Standardkonstrukten objektorientierter Sprachen wie Klassen und Methoden, als auch Spezialkonstrukten in Java, wie z.B. Assertions, abstrakte Klassen, innere Klassen, etc., welche ebenfalls einen Gültigkeitsbereich definieren und somit als Modul dargestellt werden.

Das Modul eines Blocks enthält die Sektionen: *Identifikator, Import, Körper, Export*.
Zur Erinnerung: $JPLM = \{ID, I, K, E\}$. Ein Modul wird wie folgt visualisiert:

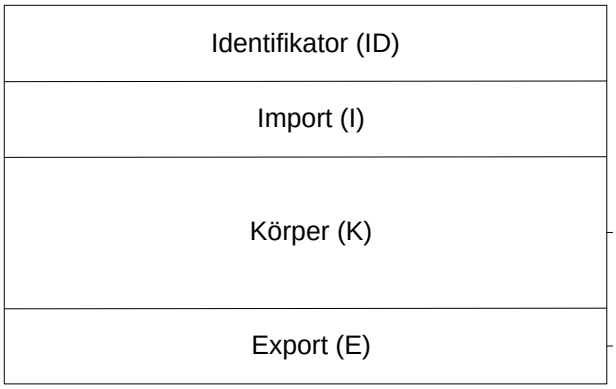


Abbildung 12: Moduldarstellung der JPL

Erläuterung der Sektionen:

Sorten: ID= $K = \{JCG \cup AJSDG \cup KaP \cup KaT\}$

Identifikation (ID): Über die Identifikator-Sektion wird festgelegt, auf welchen Gültigkeitsbereich sich das Modul bezieht. Zur Spezifikation wird ein Graph eingetragen, welcher alle Elemente des JCG enthalten darf. Allerdings ist nur ein Knoten erlaubt, der einen Gültigkeitsbereich definiert (*class, method, while*, etc.). Die in den weiteren Sektionen erstellten Elemente werden im Kontext des hierdurch gegebenen Gültigkeitsbereichs gesucht. Enthält die Identifikator-Sektion keine Elemente, so wird das Muster im vollständigen zu analysierenden Quelltexte gesucht.

Je detaillierter beschrieben wird, welcher Block untersucht werden soll, desto schneller ist die Ausführung der Mustersuche, da das Suchfeld entsprechend eingegrenzt wird. Andererseits kann durch eine sehr allgemeine Spezifikation eine größere Anzahl an Implementierungsvarianten abgedeckt werden.

Falls gefordert wird, dass zwei Module in einer hierarchischen Beziehung stehen, so können die den Gültigkeitsbereich beschreibenden Elemente mit einer *Hierarchiekante* verbunden werden.

Körper (K): Der Körper ist die Basissektion des JPL-Moduls. Hier wird die Graphstruktur spezifiziert, die innerhalb des zuvor festgelegten Gültigkeitsbereichs im Quellcode gefunden werden muss. Zur Musterbeschreibung können alle Elemente eingesetzt werden, die in Kapitel 3 dargestellt wurden.

Schnittstellensektionen

Sorten**:** $I = E = \{JCG\_SK \cup AJSDG\_SK \cup JPLSK\}$

Verwendung der Elemente des JCG_SK und JPLSK:

Import (I): In dieser Sektion werden die Strukturen der Werteübergabe beschrieben, die Variablenwerte in den Gültigkeitsbereich des Moduls importieren. Mögliche Ausprägungen sind:

1. Der Gültigkeitsbereich umfasst eine Methode und die Variable wird über deren Parameter übergeben.
2. Von einem Methodenaufruf wird ein Wert zurückgegeben.
3. Ein Wert aus einem hierarchisch höher stehendem Bereich, oder einem Bereich zu dem keine hierarchische Beziehung besteht, wird direkt referenziert (z.B. eine Membervariable, die im Rahmen einer Bedingung oder Zuweisung im Gültigkeitsbereich einer Methode verwendet wird).

Export (E): In dieser Sektion werden die Strukturen der Werteübergabe beschrieben, die Variablenwerte aus dem Gültigkeitsbereich des Moduls heraus exportieren. Mögliche Ausprägungen sind:

1. Im Gültigkeitsbereich wird eine Methode aufgerufen und Variablen werden als deren Parameter verwendet.
2. Eine Variable wird als Parameter eines *Return* Kommandos verwendet, dessen Übergabe außerhalb des im Modul betrachteten Gültigkeitsbereichs liegt.
3. Eine Variable wird an einen hierarchisch niedriger stehendem Bereich, oder einen Bereich zu dem keine hierarchische Beziehung besteht, übergeben und dort verwendet (z.B. eine Membervariable, die im Gültigkeitsbereich einer Method belegt wird).

Verwendung der Elemente des AJSDG_SK

Import (I): Ein importierter AJSDG Knoten ist nicht Teil des Gültigkeitsbereichs und ein AJSDG Knoten innerhalb des Gültigkeitsbereichs ist datenabhängig, oder kontrollabhängig von diesem.

Export (E): Ein exportierter AJSDG Knoten ist Teil des Gültigkeitsbereichs und ein AJSDG Knoten außerhalb des Gültigkeitsbereichs ist datenabhängig oder kontrollabhängig von diesem.

Die Verbindung einzelner Module über Import-/Export–Schnittstellen erfolgt mittels Datenabhängigkeitsbeziehungen, welche über Modulverbindungskanten (MVBKa) zwischen Knoten in den Schnittstellensektionen spezifiziert werden, wobei diese Elemente des JCG, des AJSDG-, als auch JPL-Schnittstellenknoten enthalten können. Es dürfen nur Elemente des gleichen Typs miteinander verbunden werden. In einem JPL Muster darf für eine Variable in einem Modul nur eine Importstruktur spezifiziert werden. Diese Importstruktur kann aus der Körpersektion beliebig oft referenziert werden (z.B. wird eine Variable als Parameter importiert und an verschiedenen Stellen im Methodenkörper referenziert). Auch der Export einer Variablen darf in einem Modul nur einmal spezifiziert werden. Zwischen Elementen im Import und im Export verschiedener Module besteht eine 1:1-Beziehung, d.h. Übergabestrukturen beinhalten immer einen Import- und einen Exportknoten, die miteinander verbunden sind.

JPL-Schnittstellenstrukturen werden im Laufe der Ableitung des abstrakten JPL-Musters in Variablenübergabestrukturen, welche nur aus JCG-Elementtypen bestehen, aufgelöst. Wird eine dieser Variablenübergabestrukturen gefunden, so wird diese dem Nutzer angezeigt.

Nicht verbundene AJSDG-Knoten in der Import- oder Exportsektion werden um Strukturen ergänzt, welche sicherstellen, dass mindestens eine der im Modul verwendeten Variablen importiert bzw. exportiert wird. Für nicht verbundene JCG Knoten werden ebenfalls im Laufe der Ableitung des Musters Strukturen zur Prüfung des Im- und Exports erstellt.

Die JPL-Moduldarstellung visualisiert den JPL-Graphen ($JPLG = \{JPLM \cup MVBKa\}$ **)**. Dieser Graph enthält alle Elemente des Moduls bzw. der Module und ihrer Beziehungen, wobei die Knoten und Kanten ein Attribut enthalten, in dem angegeben wird, in welcher Sektion das Element spezifiziert wurde.

Werden mehrere Module beschrieben, so müssen diese über mindestens eine Schnittstellenstruktur (Verbindung zwischen Import-Sektion und Export-Sektion oder über die Gültigkeitsbereiche) miteinander verbunden sein, um ein gültiges komplexes Suchmuster zu bilden. Einzelne isolierte Module sind somit in einer Gesamtstruktur nicht erlaubt.

Im Folgenden werden der Knotentyp *JPL-Schnittstellenknoten (JPLSK)* und der Kantentyp *Schnittstellenkante (SKa)* erläutert. Diese Syntaxelemente ermöglichen eine vereinfachte Spezifikation komplexer Muster, indem sie die JCG-Variablenübergabestrukturen kapseln, welche im Rahmen des Exports/Imports von Variablen aus/in den/die entsprechenden Gültigkeitsbereiche möglich sind.

4.4. Die JPL-Schnittstellenstruktur

In diesem Kapitel wird die abstrakte Schnittstellenstruktur, bestehend aus JPL-Schnittstellenknoten und der JPL-Schnittstellenkante (JPLSK und SKa) erläutert, die bei der Spezifikation der Kopplung von Modulen durch nicht genauer spezifizierte Variablenübergabebeziehungen eingesetzt wird. Da Variablenübergaben zwischen Gültigkeitsbereichen über unterschiedliche Strukturen realisiert werden können, wie z.B. durch die Übergabe eines Parameters oder den Zugriff auf eine Membervariable, müssen diese unterschiedlichen Implementierungsvarianten während der Spezifikation beachtet werden. Um den Aufwand an dieser Stelle zu verringern, werden im Folgenden die JPL-Schnittstellenknoten und –kanten eingeführt, welche diese Varianten, d.h. JCG-Graphstrukturen, im gegebenen Variablenübergabekontext kapseln. Der Typ der gefundenen Übergabestruktur wird zusätzlich zum Suchergebnis ausgegeben.

Beispiel zur Analyse der Variablenübergabe:

Anforderung:

Erstellung einer Listenverwaltung, die durch eine frei erweiterbare Verwaltungsklasse und eine vorgegebene Klasse für Listenelemente realisiert wird .

„Realisieren Sie eine Methode, die prüft, ob die Liste leer ist.“

Vorgegebenes Codefragment:

```
class Element{
Element next;
String inhalt;}

class Verwaltungsklasse{
Element listenanfang;
public boolean checkLeer(){
//vom Studenten auszufüllen
…}
```

Beide Klassen enthalten eine Deklaration vom Typ *Element*. Falls im Suchmuster zu dieser Anforderung spezifiziert wurde, dass eine Variable vom Typ *Element* in die Methode *checkLeer* importiert werden muss, so weist eine durch die Mustersuche gefundene Variablenübergabe aus einer externen Klasse in den Gültigkeitsbereich der zu implementierenden Methode wahrscheinlich auf einen Fehler hin. Es wurde möglicherweise die Deklaration des Elements *nex*t in der Klasse *Element* verwendet. Wurde jedoch eine Membervariable der Verwaltungsklasse in der Übungslösung gefunden, so ist es wahrscheinlicher, dass die Variable *listenanfang* verwendet wurde und die Lösung korrekt ist. Falls keine Analyse der Übergabevariante erfolgt wäre, würden ohne zusätzliche Spezifikation im Suchmuster beide Übergabevarianten gleichwertig als korrekt bewertet.

Durch die Angabe der Übergabevariante erhält der Nutzer eine zusätzliche Information, die er in die Bewertung des Ergebnisses der statischen Mustersuche mit einfließen lassen kann. Hierauf aufbauend kann entschieden werden, ob die Lösung weiter manuell korrigiert oder ob die automatische Auswertung als zutreffend bewertet wird. Hieraus folgt, dass durch diese Analysemöglichkeit das Risiko unerkannter *false positives* verringert wird.

JPL-Schnittstellenknoten und Schnittstellenkante

Folgende Knotentypen werden durch die Menge der JPL-Schnittstellenknoten (JPLSK) erfasst: *JPLPrimitive, JPLObject, JPLVariable, JPLSpecific*

Definition: Schnittstellenknoten werden über den Kantentyp Schnittstellenkante (SKa) miteinander in Beziehung gesetzt. Es können nur Schnittstellenknoten des gleichen Typs kombiniert werden. Die Kante spezifiziert eine 1:1 Beziehung zwischen den Schnittstellenknoten.

Schnittstellenknotentypen:
JPLPrimitive: Über den Knoten *JPLPrimitive* werden die primitiven Datentypen *int, double, long, float, char, byte, short* und *boolean* repräsentiert. Dies bedeutet, dass das Muster nur dann eine Struktur findet, wenn einer dieser Typen für die Übergabe genutzt wird.

JPLObject: Über den Knoten *JPLObject* werden von Klassen abgeleitete Objekttypen angesprochen. Das heißt, dass das Muster nur dann eine Struktur findet, wenn einer dieser Typen für die Übergabe genutzt wird.

JPLVariable: Über den Knoten *JPLVariable* werden alle Datentypen repräsentiert, d.h. es werden bei Anwendung des Musters alle Datentypen alles korrekt identifiziert.

JPLSpecific: Über den Knoten JPLSpecific können die zu suchenden Datentypen explizit angegeben werden. Hierbei muss das Attribut *DataType* entsprechend belegt werden.

Die Unterteilung in unterschiedliche Knotentypen wurde vorgenommen, da es zur Bewertung von Übungsaufgaben häufig relevant ist, welcher Datentyp im Lösungsquelltext verwendet wurde. Werden Objekte als Methodenparameter eingesetzt, so bleiben beispielsweise deren Belegungen auch außerhalb der Methode erhalten, während dies bei primitiven Datentypen nicht der Fall ist. Je nach Aufgabenkontext kann somit bereits durch die Ermittlung des verwendeten Variablentyps darauf geschlossen werden, ob ein möglicher Fehler vorliegt.

Die JPL-Schnittstellenknoten werden wie folgt verwendet:

- Ein Knoten wird im Export und ein weiterer im Import der jeweiligen Module spezifiziert. Diese werden über eine Kante vom Typ Schnittstellenkante (SKa) verbunden, die in Richtung des Importknotens gerichtet ist. Bei Anwendung der Ableitungsmethode, die pro Implementierungsvariante ein Gesamtmuster erstellt, darf aus Gründen der Musterkonsistenz nur ein JPL-Schnittstellenknoten pro Modul spezifiziert werden. Bei Anwendung der sektionsbasierten Ableitungsmethode ist diese Einschränkung nicht notwendig (zu Details der Ableitungsmethoden s. Kapitel 5.3).

- Die JPL-Schnittstellenknoten können mit Knoten vom Typ *accessToPrimitiveTypeVariable* oder *accessToReferenceTypeVariable* in der Körpersektion verbunden werden, wobei die Kantenrichtung gegen den JPL-Knoten gerichtet ist (ähnlich der Verbindung zu einem Deklarationsknoten).

Somit kann im Weiteren im Modulkörper eine Struktur spezifiziert werden, die angibt, wie die importierte und/oder exportierte Variable verwendet wird.

Die Auflösung der abstrakten JPL-Schnittstellenstrukturen in die durch diese gekapselten JCG-Graphstrukturen, wird in Kapitel 5.4 erläutert.

Im Folgenden wird beschrieben, wie Module unter Einsatz der verschiedenen in den Schnittstellensektionen zu verwendenden Knotentypen und ihren Verbindungskanten miteinander gekoppelt werden.

4.5. Modulkopplung über Datenabhängigkeiten

Die Kopplung mehrerer JPL-Module zu einer kombinierten JPL-Spezifikation wird über Verbindungen zwischen Knoten in den Schnittstellensektionen realisiert. Die Verbindung der einzelnen Module wird in der kombinierten Musterspezifikation festgelegt, indem die Knoten der Importsektion eines Moduls den Knoten der Exportsektion eines weiteren Moduls zugeordnet werden.

Kopplung über JCG Knoten:

Die folgenden JCG-Graphstrukturen realisieren grundlegende Import/Export-Strukturen zwischen zwei Modulen A und B. Die Verwendung der Deklarationsknoten erfolgt im Gegensatz zu einer reinen AST Betrachtung kontextabhängig, d.h. eine Deklaration kann in diesen Kontexten spezifiziert werden:

1. Die Variable wird in den Gültigkeitsbereich importiert (Importsektion)
2. Die Variable wird aus dem Gültigkeitsbereich heraus übergeben (Exportsektion)
3. Die Variable wird im Gültigkeitsbereich deklariert (Körpersektion)

1. Direkte Verwendung von globalen Variablen

Die Abhängigkeiten, welche über den JCG dargestellt werden, beziehen sich auf die Identifizierung einer Variablen, welche über mehrere Gültigkeitsbereiche hinweg genutzt wird. Die Identifikation dieser Variablen erfolgt über ihre Deklaration, auf welche in den verschiedenen Gültigkeitsbereichen referenziert wird.

Die im Weiteren aufgeführten Deklarationen können auf der Ebene von Klassen, Methoden, oder in Methoden geschachtelten Gültigkeitsbereichen spezifiziert werden.

Eine in Modul A deklarierte Variable wird in Modul B genutzt

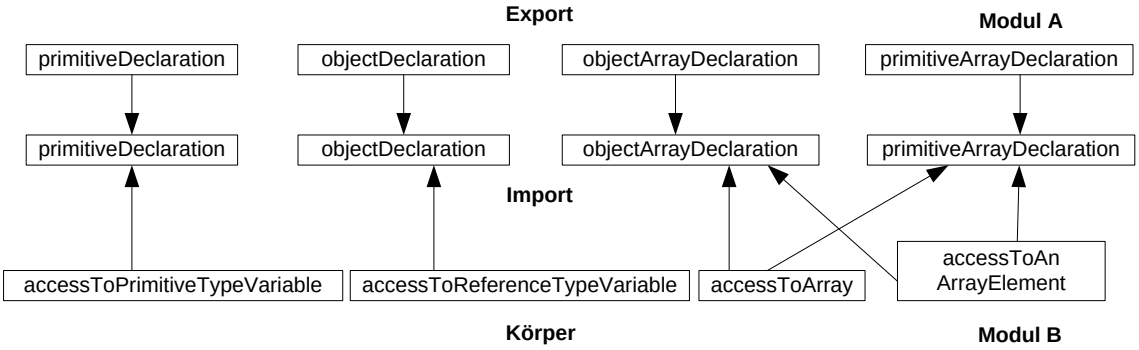


Abbildung 13: Kopplung über JCG. Direkter Zugriff auf eine Variable.

In Abbildung 13 wird sowohl die Übergabe einer Variablen eines primitiven Datentypen, als auch eines Objekts und Arrays dargestellt. Ob die Variable gelesen oder beschrieben wird, ist in dieser Struktur nicht ersichtlich.

Beispiel einer direkten Variablenbelegung:
class A{
int b;
public void uebergabe () {
b=5;
...

Die Zugriffe können sowohl im Rahmen einer hierarchischen Struktur (s. Kapitel 4.6), als auch zwischen isolierten Gültigkeitsbereichen (z.B. beim Zugriff auf die Variable einer Klasse A von Klasse B aus) erfolgen.

2. Parameterübergabe durch einen Methodenaufruf

Der Parameter eines Methodenaufrufs wird in die aufgerufene Methode exportiert.

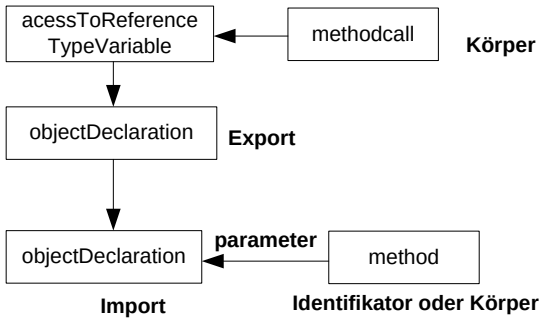


Abbildung 14: Kopplung über JCG: Indirekter Zugriff über Methodenaufruf

Während der *method*-Knoten in der Identifikator- oder Körpersektion des Moduls spezifiziert wird, wird der Parameter der Methode als Deklarationsknoten im Import aufgeführt. Das exportierende Modul spezifiziert die Deklaration des Parameters des Methodenaufrufs im Export, und den Parameter, der auf die Deklaration verweist, sowie den eigentlichen Aufruf, im Modulkörper. Im Beispiel wird eine Objektvariable eingesetzt.

3. Variablenübergabe als Return Parameter

Eine Variable einer Methode wird über das *return* Kommando in die aufrufende Methode exportiert.

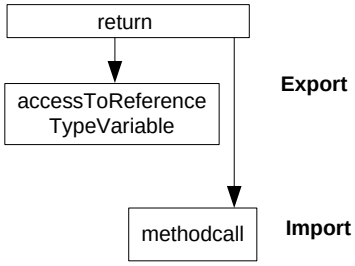


Abbildung 15: Kopplung über JCG: Indirekter Zugriff über return-Ausdruck

Der *return*-Knoten und der Rückgabewert der Methode werden im Export aufgeführt. Der Methodenaufruf, an den der Wert übergeben wird, wird im Import des Moduls beschrieben.

Kopplung über JPL-Schnittstellenknoten und -kanten

JPL-Schnittstellenknoten kapseln alle JCG-Strukturen, durch die Daten über Gültigkeitsbereiche hinweg übertragen werden können. Weiterhin wird in der JPL-Spezifikationen bei Nutzung von JPLSKs der zu suchende allgemeine Typ (Primitive, Object, Variable) der jeweiligen Variablen festgelegt, während bei Verwendung der zuvor gezeigten JCG-Elemente nur die Angabe eines konkreten Variablentyps oder keines Typs möglich ist.

Weiterhin wird bei der Ausführung der Mustersuche analysiert, aus welchem Gültigkeitsbereich im gegebenen Quelltext exportiert und in welchen Gültigkeitsbereich importiert wurde. Für die Darstellung aller erfassten JCG-Strukturen, eine Aufzählung aller Beziehungen, und deren Auflösung über Graphtransformationsregeln siehe Kapitel 5.4.

Wie in Kapitel 4.4. erklärt, werden JPL-Schnittstellenknoten wie folgt verwendet:

- Ein Knoten wird im Export und ein weiterer im Import der jeweiligen Module spezifiziert. Diese werden über eine Kante vom Typ Schnittstellenkante (SKa) verbunden, die in Richtung des Importknotens gerichtet ist. Bei Anwendung der Ableitungsmethode, die pro Implementierungsvariante ein Gesamtmuster erstellt, darf aus Gründen der Musterkonsistenz nur ein JPL-Schnittstellenknoten pro Modul spezifiziert werden. Bei Anwendung der sektionsbasierten Ableitungsmethode ist diese Einschränkung nicht notwendig (zu Details der Ableitungsmethoden s. Kapitel 5.3.).
- Die JPL-Schnittstellenknoten können mit Knoten vom Typ accessToPrimitiveTypeVariable oder accessToReferenceTypeVaribale in der Körpersektion verbunden werden, wobei die Kantenrichtung gegen den JPL-Knoten gerichtet ist (ähnlich der Verbindung zu einem Deklarationsknoten). Somit kann im Weiteren im Modulkörper eine Struktur spezifiziert werden, die angibt, wie die importierte und/oder exportierte Variable verwendet wird.

Im folgenden Beispiel werden die Kopplung zweier Gültigkeitsbereiche als JPL-Graph und zwei Quelltextfragmente, welche über diese Spezifikation gefunden würden, dargestellt:

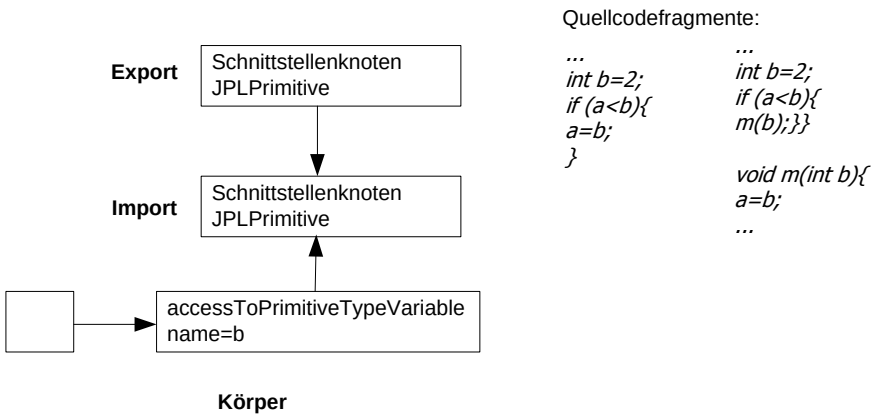


Abbildung 16: Beispiel für die Verwendung von JPL-Schnittstellenknoten

Durch das Suchmuster in Abbildung 16 wird im linken Quellcode die Übergabe der Variablen *b* in den Gültigkeitsbereich der *if*-Anweisung erfasst. Hierbei handelt es sich um eine Übergabe im Rahmen einer hierarchischen Beziehung (für Erläuterungen zu hierarchischen Strukturen siehe nächstes Kapitel).
Im rechten Quelltext wird zum einen die Übergabe von *b* in den Körper der *if*-Anweisung und die dortige Verwendung als Methodenparameter identifiziert. Zum anderen wird die Parameterübergabe in den Gültigkeitsbereich der Methode *m* gefunden, d.h. im rechten Quellcode existieren zwei Matches für die links dargestellte Spezifikation.

Kopplung über AJSDG-Knoten:

Während bei der Kopplung durch den JCG eine einzelne Variable betrachtet wird, werden über den AJSDG vollständige Ausdrücke spezifiziert. Der Im- und Export erfolgt hier für Datenabhängigkeiten, d.h. es werden die Anweisungsknoten exportiert, welche den Wert einer Variablen eines anderen Blocks beeinflussen. Bereits bestehende AJSDG-Beschreibungen, welche in der Körpersektion des Musters spezifiziert wurden, können mittels dieser Spezifikation modulübergreifend erweitert werden. Die Vorteile welche sich durch eine Abstraktionsebene über dem JCG ergeben, wurden bereits in Kapitel 3.3 dargelegt.

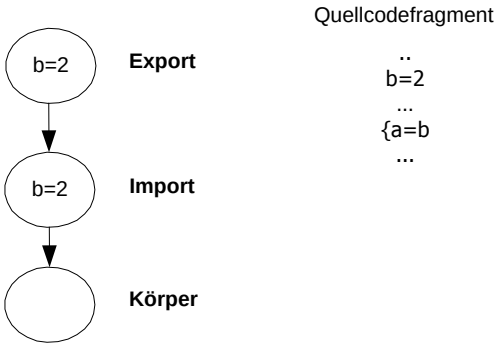


Abbildung 17: Kopplung über AJSDG-Knoten

In Abbildung 17 wird eine Datenabhängigkeit zwischen zwei Modulen dargestellt. Hierdurch wird spezifiziert, dass im importierenden Modul eine Anweisung datenabhängig von der Anweisung *b=2* des exportierenden Moduls ist. Im Beispiel ist die Anweisung *a=b* diese abhängige Struktur, welche durch das Muster gefunden würde. Im Folgenden wird erläutert, wie hierarchische Beziehungen in der JPL realisiert werden.

4.6. Repräsentation hierarchischer Strukturen

Gültigkeitsbereiche im Programmcode sind hierarchisch geschachtelte Strukturen. Der übergeordnete Gültigkeitsbereich umfasst die in ihm eingebetteten Bereiche und steht somit hierarchisch über diesen, so steht eine Klasse z.B. hierarchisch über einer Methode. In der JPL können diese Strukturen in Suchmustern mit mehreren Modulen explizit gefordert werden.

Implizite Erfassung von Hierarchieebenen in einem einzelnen Modul:

Suchmuster die in einem Modul definiert sind, beziehen sich sowohl auf die Ebene des in der Identifikatorsektion definierten Bereichs als auch auf alle eingebetteten Gültigkeitsbereiche. Falls ein Modul z.B. eine Klasse als Gültigkeitsbereich definiert, so dürfen die Elemente des Suchmusters auch in den Methoden, Schleifen, etc. der gefundenen Klasse enthalten sein, um als Muster erkannt zu werden.

Explizite Spezifikation einer hierarchischen Beziehung zwischen zwei Modulen:

Hierarchien zwischen Einzelmodulen können explizit spezifiziert werden. So ist es z.B. möglich festzulegen, dass eine Methode in einer zuvor spezifizierten Klasse enthalten sein muss (s. Abbildung 18). Zur Festlegung der Hierarchieebenen wird ein Modul (A) für den umfassenden Gültigkeitsbereich und ein Modul (B) für den eingebetteten Gültigkeitsbereich erstellt. Die Hierarchiebeziehung wird über eine Hierarchiekante zwischen beiden Modulen spezifiziert. Die Kante verbindet die Identifikatoren beider Module, wobei die Kantenrichtung vom umfassenden zum eingebetteten Block zeigt.

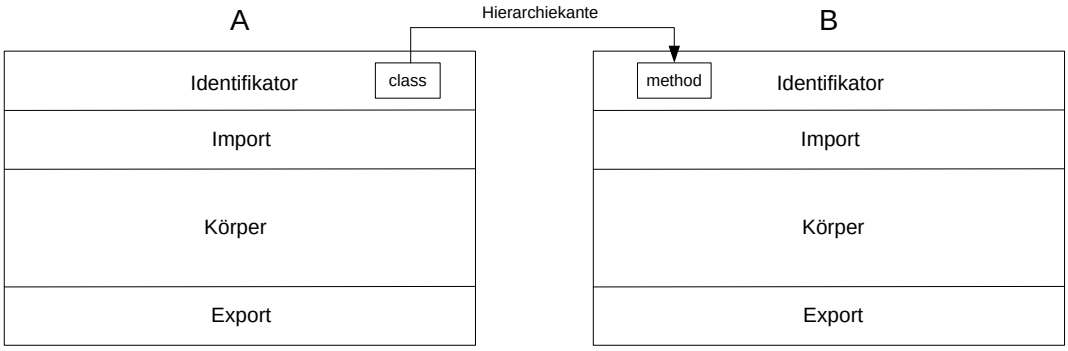


Abbildung 18: Hierarchiebeziehungen in der JPL

Die oben dargestellte Vorgehensweise kann iterativ über mehrere Blöcke hinweg angewendet werden. So könnte in dem oben spezifizierten Gültigkeitsbereich des Moduls der Methode noch ein weiterer Block, z.B. einer Schleife, definiert werden. Das

Beispiel würde dann um ein Modul ergänzt, welches den JCG-Knoten vom Typ *while* in der Identifikatorsektion enthält. Dieser Knoten wäre mit dem entsprechenden *method*-Knoten des Moduls B über eine Hierarchiekante verbunden. Alternativ kann die Schleife im Körper der Methode spezifiziert werden.

Im Folgenden wird die Kopplung von Modulen für Strukturen betrachtet, welche mehrere semantisch zusammenhängende Gültigkeitsbereiche besitzen. Die Kopplung erfolgt, ähnlich zu der Kopplung in expliziten Hierarchiebeziehungen, über die Identifikator-Sektion.

4.7. Kopplung semantisch zusammenhängender Module

In der Programmiersprache Java existieren Programmierkonstrukte, welche eine enge semantische Bindung zwischen zwei Gültigkeitsbereichen definieren. Hierbei können zwei Typen unterschieden werden:

- Bedingungen: In diesen Strukturen wird über eine Bedingung am Strukturanfang entschieden, welcher Gültigkeitsbereich zur Laufzeit erreicht wird. Diese Konstrukte sind: *if...then...else* und *switch…case.*
- Fehlerbehandlungen: Bei Eintritt eines Fehlerfalls wird in den verbundenen zweiten Gültigkeitsbereich der Struktur gesprungen. Diese Konstrukte sind: *try…catch, assert…catch*.

Die Kopplung von semantisch zusammenhängenden Gültigkeitsbereichen beider Kategorien erfolgt über die Identifikator-Sektion. Die JCG-Knoten werden über die jeweils zugehörige JCG-Kante verbunden. Abbildung 19 zeigt die Kopplung im Rahmen einer if-Bedingung.

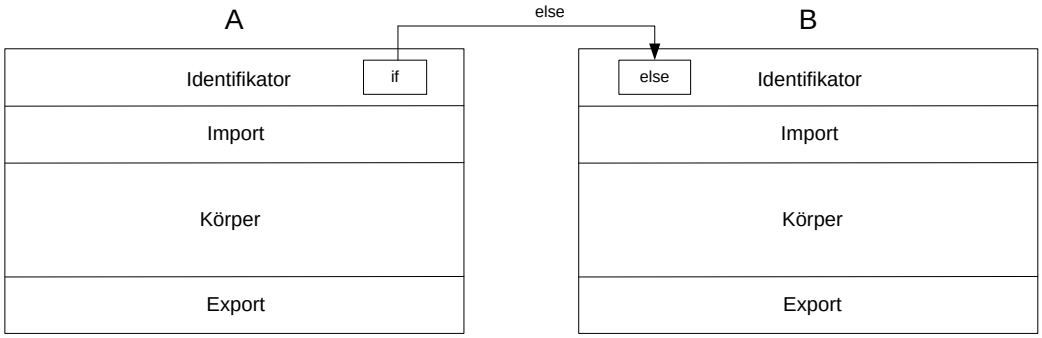


Abbildung 19: Repräsentation einer *if...then* Bedingung

Weitere Kopplungen werden über folgende JCG-Kantentypen realisiert:
- MVBKaS={else, if, case, default, catch}
- Switch-Bedingung: von: *switch* nach *case*, oder *default*.
- If-Bedingung: von *if* nach *else*, oder *elseif.*
- Try-Struktur und Assert-Struktur: von *try* oder *assert* nach *catch.*

Nachdem in den vorhergehenden Kapiteln alle Elemente der JPL definiert und erläutert wurden, folgen nun Beispiele, welche deren Einsatz verdeutlichen.

4.8. Beispiel zur Kopplung über JCG-Elemente

Im diesem Abschnitt werden beispielhaft komplexe Musters mit Modulverbindungen über JCG Strukturen dargestellt.

Anforderung:

„Belegen Sie in einer Methode eine Variable und geben Sie diese in einer weiteren Methode aus.“

Lösungsvariante 1:

```
class {
String  i;
public void m1(){
        Systems.out.println(i);
}
public void m2 (){
        i=System.in();
        m1();
}}
```

Lösungsvariante 2:

```
class {
public void m1(String u){
        System.out.println(u);
}
public void m2 (){
        String a;
        a=System.in();
        m1(a);
}}
```

Um beide Lösungsvarianten abzudecken, müssen zwei unterschiedliche JCG-Mustervarianten erzeugt werden.

Suchmuster für Variante 1:

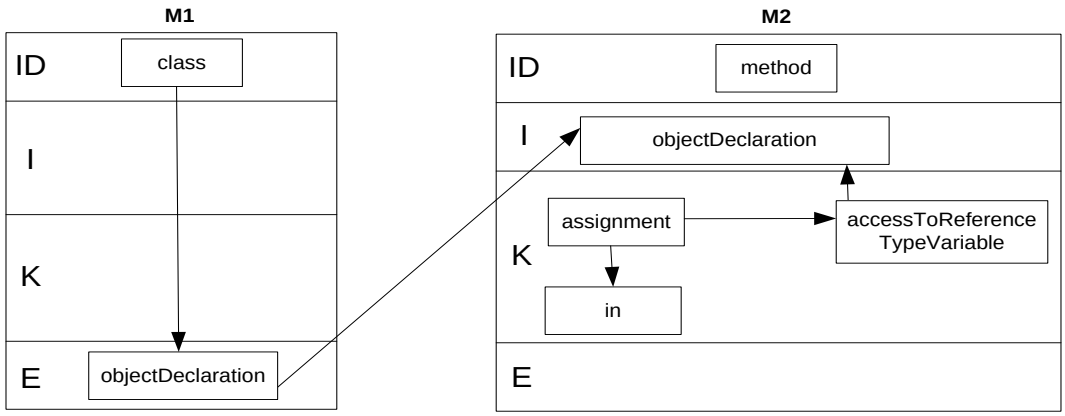


Abbildung 20: Suchmuster für Variante 1: Belegen der Variablen

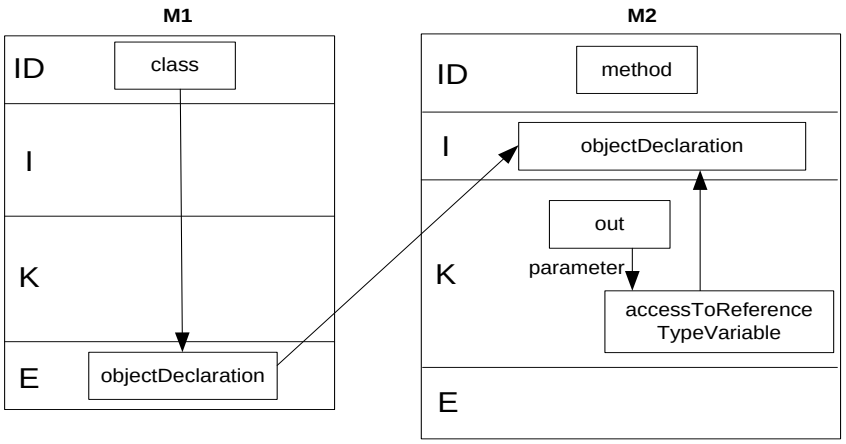


Abbildung 21: Suchmuster zu Variante 1: Ausgabe der Variablen

Das Muster in Abbildung 20 spezifiziert, wie eine Membervariable belegt wird. Modul 1 wird zur Identifizierung einer Membervariablen genutzt, während in Modul 2 die Variable unter Verwendung der *in*-Methode belegt wird. Das Muster in Abbildung 21 beschreibt das Auslesen einer Membervariablen. Über Modul 1 wird die Membervariable identifiziert und in Modul 2 wird der Wert ausgegeben. Zur besseren Übersichtlichkeit wurden verschiedene Elemente, wie z.B. der Chainingknoten vor den Methoden *in* und *out*, in der Abbildung nicht dargestellt. Zu beachten ist, dass durch die beiden Muster nur spezifiziert wird, dass eine Variable beschrieben und eine Variable ausgelesen wird. Es wurde nicht spezifiziert, dass durch beide Muster die gleiche Variable identifiziert werden muss oder dass diese zuerst belegt und dann ausgelesen wird. Der dynamische Ablauf kann hieraus nicht abgeleitet werden. Die Übergabemethode wurde in Kapitel 4.5 Abbildung 13 dargestellt.

Suchmuster für Variante 2:

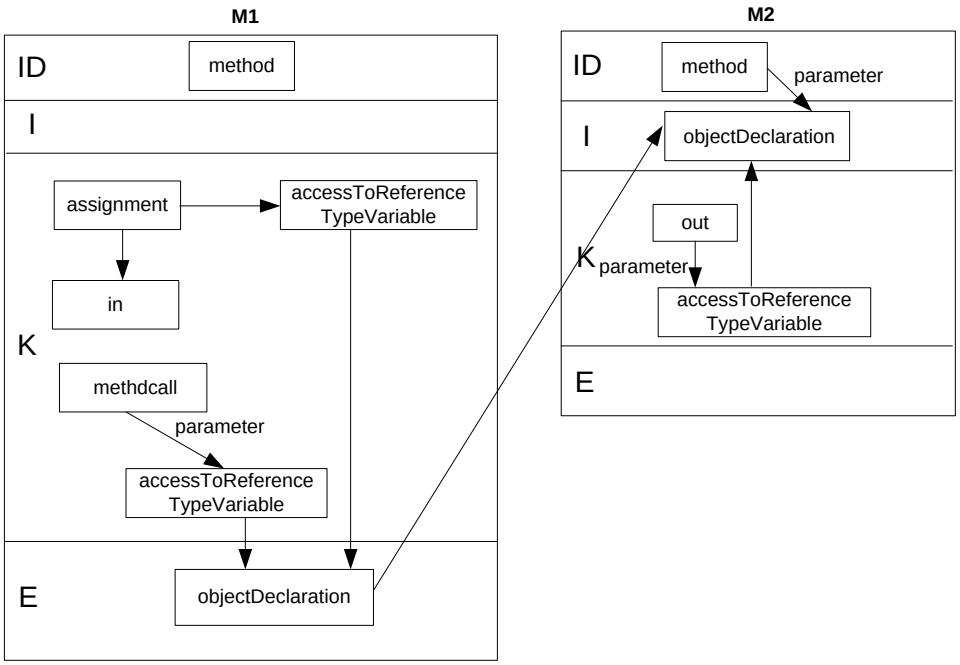


Abbildung 22: Suchmuster zu Variante 2: Belegen und Ausgabe der Variablen

In der Suchmustervariante 2 wird ein Objekt mit dem Wert der Methode *in* belegt. Das Objekt wird als Parameter eines Methodenaufrufs an eine weitere Methode übergeben. Hier wird der Inhalt der Variablen durch die Methode *out* ausgegeben. Die Übergabemethode wurde in Kapitel 4.5., Abbildung 14 dargestellt.

Bei Verwendung einer abstrakten Schnittstellenstruktur ist nur ein Muster notwendig, welches beide Implementierungsvarianten erkennt.

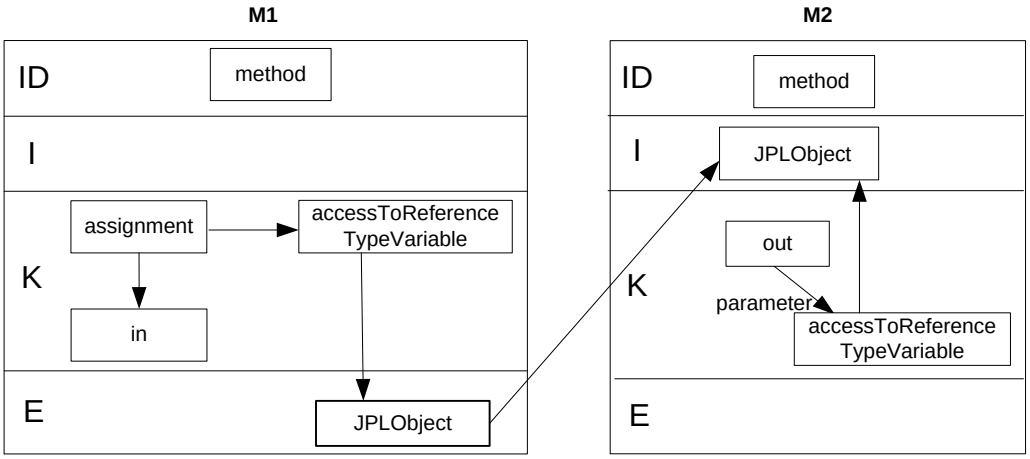


Abbildung 23: Suchmuster unter Verwendung von JPL-Schnittstellenknoten

In Modul 1 wird spezifiziert, dass ein Objekt durch die Methode *in* belegt wird. Dieses Objekt wird von einer weiteren Methode ausgelesen und über die Methode *out* ausgegeben. Die Art der Übergabe ist nicht genauer spezifiziert, so dass hier im Rahmen der Ableitung der JCG-Suchmuster aus den JPL-Schnittstellenknoten alle Möglichkeiten, wie z.B. die Übergabe über eine Membervariable (hierbei wird im Gegensatz zur Verwendung der beiden Suchmuster in Variante 1 sichergestellt, dass die gleiche Variable verwendet wird) oder als Parameter, geprüft werden. Nachfolgend wird angezeigt, welche Variante in der zu analysierenden Lösung verwendet wurde.

Im folgenden Abschnitt wird die Nutzung der AJSDG-Knoten am Beispiel näher erläutert.

4.9. Beispiel zur Kopplung über AJSDG-Elemente

In diesem Kapitel wird beispielhaft beschrieben, wie Beziehungen zwischen Modulen in komplexen Strukturen über AJSDG-Knoten erstellt werden.

Anforderung:
„Schreiben Sie eine Methode *insert*, welche Elemente der Klasse *Student* in der Listenstruktur *Studentenliste* einfügt. Der Name des Studenten wird an die Methode übergeben."

Nutzen Sie dazu folgende Vorlagen:

```
public class Student{
String name;
Student next;
Student (String name){… }
}

public class Studentenliste{
Student head;
public void insert (…)…
}
```

Folgende Teilanforderung wird durch das Suchmuster in Abbildung 24 erfasst: Der Name des Studenten wird an die Methode übergeben.

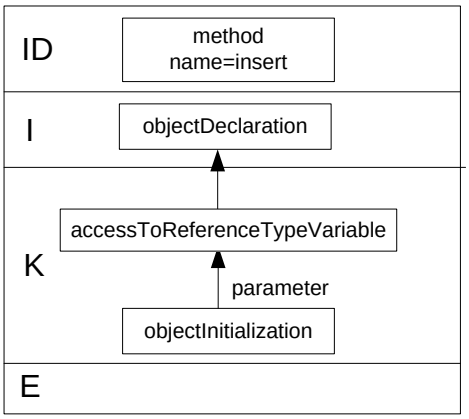


Abbildung 24: JPL-Spezifikation zu Anforderung 1

Es ist zu beachten, dass im Muster nicht nur nach einem importierten Objekt gesucht wird, sondern die Referenz auf dieses Objekt auch im Rahmen einer Objektinstanziierung genutzt wird.

Das zweite Muster beschreibt eine mögliche Implementierungsvariante der Anforderung: "Der Parameter des Konstruktors wird zur Belegung der Membervariablen *name* in der Klasse Student genutzt."

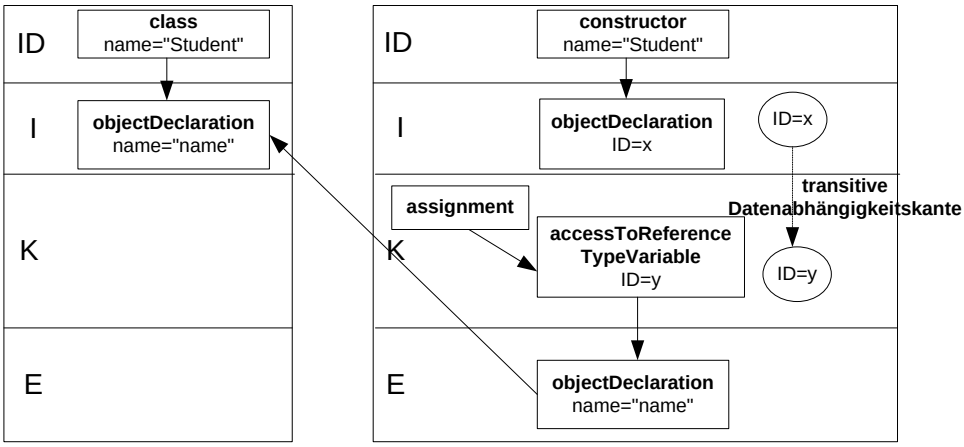


Abbildung 25: JPL-Spezifikation zu Anforderung 2

Abbildung 25 zeigt die Spezifikation der oben genannten Anforderung. Es ist zu beachten, dass der Name über den Konstruktorparameter importiert, und nachfolgend durch die Belegung der Membervariablen *name* exportiert werden muss. Durch die Spezifikation der Datenabhängigkeit der Referenzierung der Membervariablen *name* der Klasse *Student* im Konstruktorkörper vom Parameter des Konstruktors, wird die Belegung der Membervariablen durch diesen Parameter beschrieben. Die transitive Kante deckt die Möglichkeit ab, dass Hilfsvariablen implementiert werden, welche den Wert zwischenspeichern.

Das dritte Muster kombiniert die zuvor gezeigten Muster, so dass nun folgende Anforderung geprüft wird: „Der Wert, welcher in die Methode *insert* importiert und im Rahmen der Instanziierung des Studentenobjekts als Konstruktorparameter verwendet wird, wird der Variablen *name* der Klasse *Student* zugeordnet.“

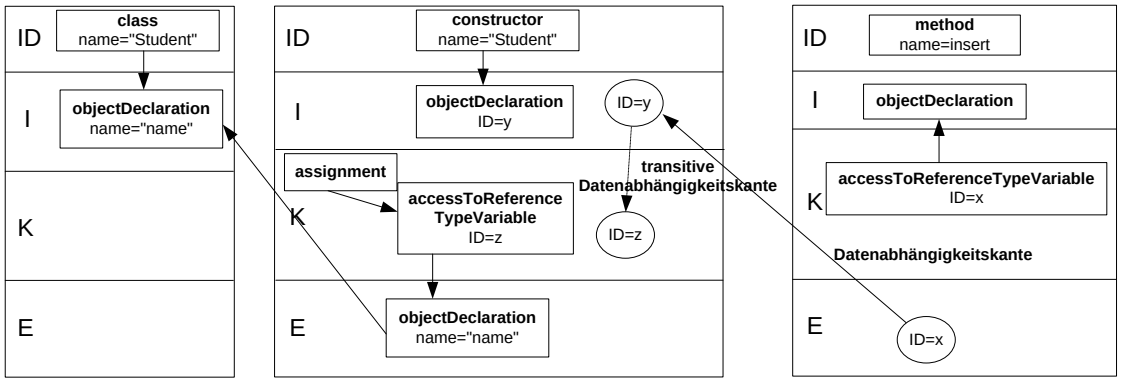


Abbildung 26: JPL-Spezifikation zu Anforderung 3

Die Verbindung wird durch eine Datenabhängigkeitsbeziehung hergestellt, welche sich auf den Konstruktorparameter in der Methode *insert* bezieht. Da eine Datenabhängigkeit zwischen dem Parameterknoten mit den IDs x und y besteht, existiert auch eine Datenabhängigkeit zwischen den Knoten x und z, so dass die komplexe Anforderung über mehrere Gültigkeitsbereiche durch dieses Muster erfasst wird. Da ein Konstruktor immer über eine Instanziierung aufgerufen wird, konnte der entsprechende JCG-Knoten im rechten Modul ausgelassen werden. Dies bedeutet, dass

die Spezifikation des rechten Moduls unabhängig davon erfolgen kann, wie der Name an das Studentenobjekt übergeben wird.

In diesem Abschnitt wurde ein Einsatz der AJSDG-Elemente zusammen mit JCG-Elementen erläutert, um die theoretischen Beschreibung der Verwendung der Elemente in den vorhergehenden Abschnitten durch ein Beispiel zu verdeutlichen.

4.10. Zusammenfassung

Durch die in diesem Kapitel dargestellte Musterspezifikationssprache JPL werden folgende in Abschnitt 1.2. definierten Anforderungen erfüllt:

- Das Suchmuster wird visuell über eine modularisierte Graphstruktur beschrieben.
- Die Erstellung konsistenter Einzelmuster wird durch die Spezifikation mittels Elementen des JCG und AJSDG unterstützt. Konsistenz bezieht sich hier auf die Relation zwischen der durch eine Anforderung geforderte Struktur, und dem hierzu erstellten Suchmuster, das diese Anforderung abbildet. Besonders die Syntaxelemente, welche implizite Beziehungen im Quelltext explizit graphisch darstellen, können in diesem Zusammenhang zur Konsistenzsicherung beitragen. Hier sind z.B. die Beziehung zwischen Variablenverwendung und Deklaration oder die Darstellung der Kontrollabhängigkeiten zu nennen. Diese Aussage basiert auf der Annahme, dass ein Nutzer Beziehungen zwischen Elementen visuell einfacher erfassen und spezifizieren kann, als textuell z.B. über Attribute der jeweiligen Knoten.
- Durch die Verwendung von Schnittstellen können komplexe Suchmuster aus Einzelmustern strukturiert zusammengesetzt werden. **Realisierung:** Ein einzelnes Muster bezieht sich immer auf einen Gültigkeitsbereich. Der Im- und Export von Variablen in/aus diesen Gültigkeitsbereiche/n wird in den entsprechenden Sektionen festgelegt, so dass mehrere Muster über diese Schnittstellen konsistent miteinander verbunden werden können. Weiterhin können Hierarchiebeziehungen über die Identifikatorsektion spezifiziert werden.
- Der Spezifikationsaufwandes für komplexe Muster wird durch Abstraktionen verringert, welche nachfolgend automatisch detailliert werden, um *false positives* zu vermeiden:

  Bei der Verwendung von JPL-Schnittstellenknoten in den Import/Export-Sektionen werden nur Variablen mit ihrem Variablentyp angegeben, aber es wird nicht definiert, wie die Übergabe implementiert werden muss. So werden mehrere Implementierungsvarianten in einer Musterdefinition abstrakt erfasst. **Realisierung:** Es wird von der konkreten Ausprägung der Datenübergabe zwischen den in den Modulen erfassten Gültigkeitsbereichen abstrahiert, d.h. die Implementierung der Übergabe wird bei der Beschreibung der Modulbeziehungen durch den Nutzer der Sprache nicht näher betrachtet. Die Strukturen der Implementierungsvarianten werden nachfolgend automatisch durch die entsprechenden Graphregeln erzeugt (s. folgendes Kapitel). Da auch Übergaben durch globale Variablen erfasst werden, kann im Rahmen einer statischen Analyse nicht erkannt werden, ob diese Übergabe während des

Ablaufs des Programms auch erfolgt. Es wird lediglich eine Struktur identifiziert, welche eine Variable v im Gültigkeitsbereich a belegt und in Bereich b ausliest.

- Da es dem Nutzer überlassen bleibt, wie detailliert er den Gültigkeitsbereich des Moduls spezifiziert, werden durch ein Modul verschiedenste Implementierungsvarianten erfasst. **Realisierung:** In der Sektion *Identifikator* kann angegeben werden, in welchem Gültigkeitsbereich das in diesem Modul spezifizierte Muster gesucht werden soll. Bei entsprechend abstrakter Definition des Identifikators kann das Muster in den verschiedensten Gültigkeitsbereichen enthalten sein. Die Nachteile dieser Vorgehensweise liegen im größeren zu analysierenden Suchraum, und einer entsprechend schlechten Performance der Suche, sowie eines möglicherweise größeren Risikos von *false positives*. Dies wird durch eine engere Eingrenzung des Suchraums vermieden, wodurch allerdings das Risiko von *false negatives* ansteigt.

In Kapitel 6 werden grundlegende Strukturen, die im Rahmen der Mustersuche eingesetzt werden können, hinsichtlich ihrer Eignung analysiert. Kriterien sind hier die Qualität der gefundenen Lösungsstrukturen und der Performance der Mustersuche. Eine Analyse, für welche Anforderungen die JPL besonders gut geeignet ist, und in welchen Kontexten sie nicht eingesetzt werden sollte, wird in Kapitel 8.5. im Anschluss an die Betrachtung praktischer Anwendungsbeispiele gegeben.

Im folgenden Kapitel wird die Ableitung des abstrakten JPL-Musters zu einem oder mehreren Graphmustern, welche nur Elemente der Typen AJSDG und JCG enthalten, beschrieben. Diese Muster können nachfolgend direkt auf dem Quelltext gesucht werden. Die Transformation erfolgt über die Methode der Graphtransformation, deren nähere Betrachtung den Einstieg in das folgende Kapitel bildet.

5. Ableitung des JPL-Graphen und Durchführung der Mustersuche

In diesem Kapitel wird die Ableitung des modular aufgebauten JPL-Musters in mehrere nicht modulare Graphstrukturen, die ausschließlich aus JCG und AJSDG Elementen bestehen und somit direkt auf die Graphdarstellung des zu untersuchenden Quellcodes angewendet werden können, dargestellt. Diese Graphen repräsentieren Muster für die verschiedenen Implementierungsvarianten, welche im modularen JPL-Suchmuster gekapselt sind. Die Transformation erfolgt über Graphtransformationsregeln, wobei der algebraische Graphtransformationsansatz nach [EEP+06] angewendet wird. Weiterhin wird der Ablauf der Mustersuche erläutert, über den die verschiedenen Implementierungsvarianten identifiziert werden können.

Zu Anfang dieses Kapitels wird der konstituierende Hintergrund des Graphtransformationsansatzes dargestellt, der in den folgenden Abschnitten zur Musterableitung und der Durchführung der Mustersuche verwendet wird. In Abschnitt 5.3. wird beschrieben, wie ein abstraktes JPL-Muster, bzw. dessen einzelne Sektionen und Elemente, in die verschiedenen konkreten Mustervarianten überführt werden. Zwei grundlegende Vorgehensweisen können unterschieden werden. Zum einen kann das spezifizierte Gesamtmuster in die verschiedenen Mustervarianten transformiert werden, welche nachfolgend gesucht werden. Zum anderen können die Strukturen der verschiedenen Module und Sektionen separat betrachtet werden, wobei hier für den Ablauf der Mustersuche ein Prozess erstellt werden muss, über den das Gesamtmuster auf Grund der Suchergebnisse der verschiedenen Teilmuster identifiziert wird. Die Vor- und Nachteile der beiden Varianten im Rahmen der Suchmusterspezifikation und des Suchablaufs werden detailliert in Kapitel 5.3. aufgeführt. Nachdem die für beide Varianten notwendigen allgemeinen Ableitungsschritte beschrieben wurden, erfolgt ab Kapitel 5.4. die Darstellung der Regelmenge, welche JPL-Schnittstellenstrukturen in die verschiedenen konkreten Strukturen zur Variablenübergabe auflöst, von denen im JPL-Muster abstrahiert wurde.

Anschließend werden in Kapitel 5.8. die Graphregeln erläutert, welche die Transitive Hülle auf dem Quellcodegraph erstellen. Die Beschreibung wurde in diesem Kapitel platziert, da die Erstellung der Transitiven Hülle über die Technik der Graphtransformation erfolgt, die in Kapitel 5 eingeführt wird. Die weiteren in Kapitel 3 dargestellten Syntaxelemente der JPL werden entweder direkt aus dem AST generiert (JCG), oder über Graphtransformationsregeln erzeugt, die in bereits bestehenden Arbeiten beschrieben werden (JSDG) [WH07].

Im folgenden Kapitel wird der Ablauf der Mustersuche erläutert und an einem Beispiel verdeutlicht. Der Algorithmus zur Generierung der notwendigen Ablaufdatei wird in den Kapiteln 5.9. und 6 beschrieben. Nachfolgend kann die Mustersuche auf dem Graphen des zu analysierenden Quellcodes erfolgen. Das Resultat der Mustersuche sind Ergebnisknoten, welche z.B. den Text eines Fehlerfalls, gefundene Variablenübergabevarianten, oder die allgemeine Bestätigung eines Musterfundes beinhalten können.

Die folgende Abbildung zeigt den Prozess von der Erstellung des JPL-Musters bis zur Ausführung der Mustersuche.

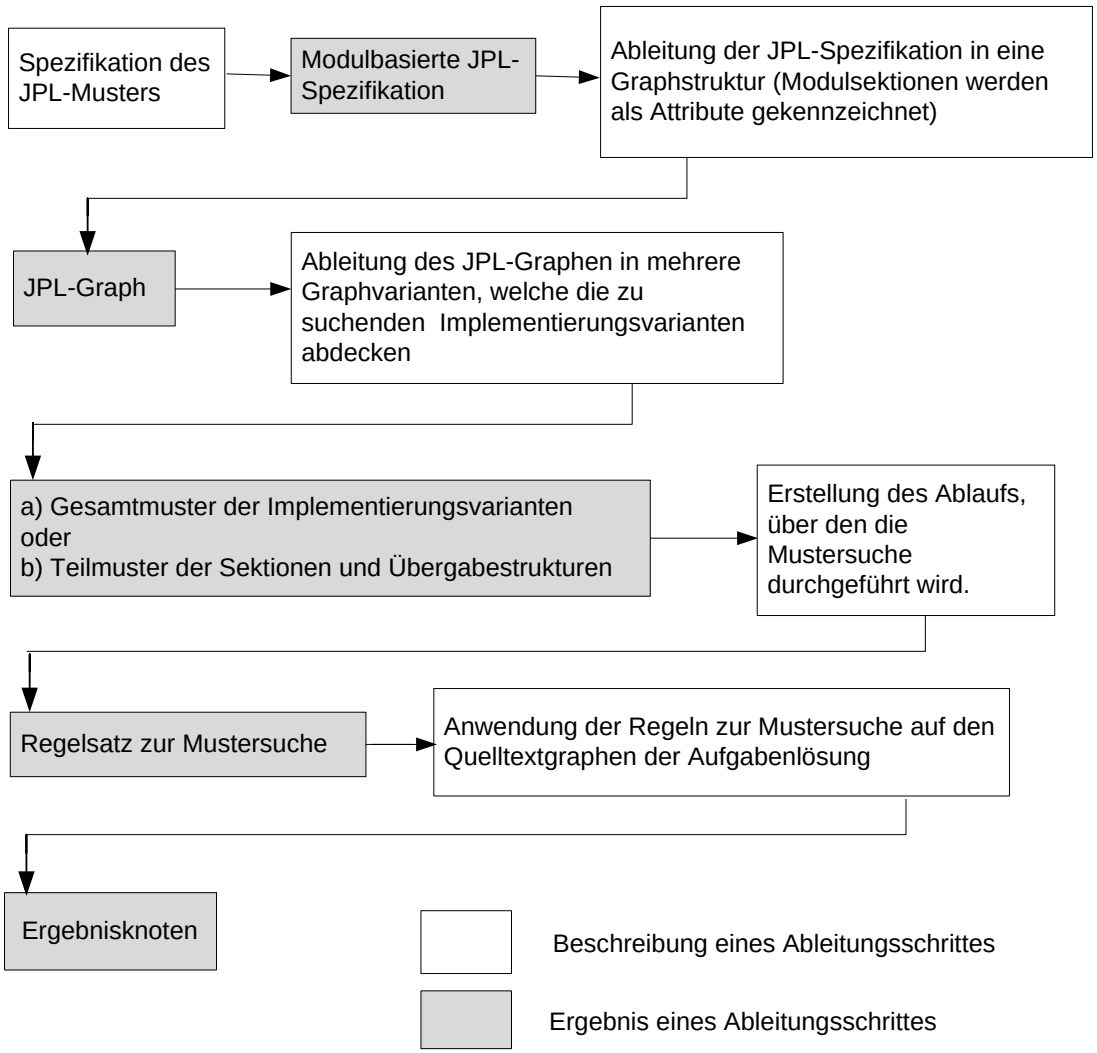


Abbildung 27: Prozess der Ableitung des JPL-Graphen in Graphtransformationsregeln sowie Ausführung der Mustersuche

Abschließend wird die Technik des Slicing und deren Ausführung über Graphtransformationsregeln für die in dieser Arbeit eingeführte Quelltextrepräsentation dargestellt. Diese Technik wird in Kapitel 6 zur Realisierung einer Methode für die Erstellung von JPL-Spezifikationen eingesetzt.

Die nachfolgenden beiden Kapitel führen in die Modifikation von Graphen über die Technik der Graphtransformation ein. Es werden sowohl verschiedene Anwendungsgebiete betrachtet, als auch argumentiert, warum die Technik im Kontext dieser Arbeit verwendet wird. Die Erläuterungen legen die Grundlage für die Betrachtungen der weiteren Unterkapitel.

5.1. Graphtransformation im Software Engineering

Die Technik der Graphtransformation [Roz97] beschreibt eine regelbasierte Veränderung von Graphstrukturen. Diese besteht aus zwei grundlegenden Teilschritten: 1) Identifikation einer spezifizierten Struktur, nachfolgend *Match* genannt und 2) die Transformation der gefundenen Struktur durch das Hinzufügen, Löschen, oder Verändern von Elementen. Die Konzepte und die formale Definition der in dieser Arbeit genutzten Graphtransformationstechnik basieren auf dem „Double-Pushout Ansatz“, welcher auf die algebraische Graphtransformation angewendet wird. Die im Folgenden verwendete algebraische Graphtransformation wird in [CMR+97] und [ERT99] eingeführt. Hierbei wird der ursprüngliche Graphtransformationsansatz um Vorbedingungen (Negative Application Conditions) und Attribute ergänzt, welche sowohl auf Knoten als auch auf Kanten angewendet werden können.

Im Rahmen dieser Arbeit wird die Methode der Graphtransformation zum einen für die Transformation eines modularisierten JPL-Suchmusters in die Darstellung eines „flachen“ nicht modularen Graphen, der ausschließlich aus JCG und AJSDG Elementen besteht, verwendet, und zum anderen, um die einzelnen Schritte der Suche des Musters über dem Quellcodegraphen auszuführen.

Die Eignung der Graphtransformation als Methode für Analysen im Rahmen von Softwarearchitektur- und Softwaredesignfragestellungen wurde bereits in einer Vielzahl von Arbeiten dargelegt. Verschiedene Forschungsarbeiten beschreiben die Struktur und das Verhalten von Softwaresystemen mittels des Ansatzes der algebraischen Graphtransformation. So wurde die Graphtransformation bereits zur Beschreibung von Zustandsdiagrammen und Petrinetzstrukturen eingesetzt. Ein allgemeiner Ansatz zur Veränderung von UML-Modellen wird in [JJA06] und [KKL+07] gegeben, wobei hier Graphregeln als Aspekte definiert werden, welche in ein bestehendes Diagramm eingefügt werden.

Anwendungsgebiete von Graphtransformationstechniken auf Quellcode-Ebene sind sowohl die Realisierung von Refaktorings [EJ04], [MTR06], als auch die Ableitung von UML-Diagrammen zu Quellcode [Fuj14], welche in umfangreichen Forschungsbeiträgen veröffentlicht wurden. Hierbei wird sowohl die Erkennung von Pattern, als auch die Analyse ihrer Abhängigkeiten im Rahmen von Graphtransformationsregeln realisiert. Eine kurze Beschreibung, inwieweit diese Techniken mit der hier betrachteten Arbeit kombiniert, bzw. ergänzt werden könnten, wird in Kapitel 9.3. gegeben.

Die oben aufgeführten Ansätze kombinieren die intuitive Verwendung von Graphen mit einer formal fundierten Basis. Die Darstellung und Transformation realistischer Softwaresysteme über Graphen wird allerdings schnell unübersichtlich, so dass es notwendig ist Abstraktionsebenen einzuführen. Zwei grundlegende Ansätze können hier im Kontext der Graphtransformationsregeln unterschieden werden. In [KK96] werden Graphregeln zu Units zusammengefasst und gemeinsam in einem Set betrachtet. Dieser Ansatz wurde im Graphtransformationstool AGG [agg14] aufgegriffen und über die Anwendung von Layern realisiert. Der zweite Ansatz [EE96] realisiert die Modularisierung von Graphtransformationsregeln über algebraische Signaturen und über die hiermit gekoppelten Beziehungen zwischen Regeln.

Rechtfertigung der Wahl des algebraischen Graphtransformationsansatzes

Neben der algebraischen Graphtransformation existieren noch weitere Graphtransformationsansätze, deren Einsatz im Kontext dieser Arbeit betrachtet wurde. Zu nennen sind hier der mengentheoretische Ansatz und der logikorientierte Ansatz [FMR+07].

Im mengentheoretischen Graphtransformationsansatz wird ein Graph als Menge von Knoten und Kanten beschrieben. Eine Transformationsregel wird als mengentheoretische Operation formuliert. Kanten werden im Gegensatz zum algebraischen Ansatz lediglich als Relation zwischen Knoten aufgefasst, so dass diese nicht identifizierbar oder attributierbar sind. Weiterhin werden keine allgemeinen Grapheigenschaften einbezogen, über die globale Bedingungen formuliert werden könnten.

Im logikbasierten Graphtransformationsansatz, welcher den mengenorientierten Ansatz erweitert, sind die explizite Kantendarstellung und die Globalbetrachtung in das Konzept integriert worden, wobei Graphen und Graphtransformationsregeln über Ausdrücke der Prädikatenlogik formuliert werden. [Roz97].

Im Gegensatz zu den oben genannten Ansätzen definiert die algebraische Graphtransformation einen Graphen als algebraische Sprache mit zwei grundlegenden Typen: Knoten und Kanten, wobei jedes Element einem Typ zugeordnet werden kann sowie identifizierbar und attributierbar ist. Hierdurch ist gegenüber dem mengentheoretischen Ansatz eine wesentlich höhere Mächtigkeit in der Definition eines Graphen gegeben, welche für die in dieser Arbeit eingesetzten komplexen Transformationen unerlässlich ist. Transformationen der algebraischen Graphtransformation werden als *Pushouts* definiert, welche im für diese Arbeit gegebenen Kontext der visuellen Spezifikation von Suchmustern intuitiver eingesetzt werden können, als die Definition von Mustern über Formeln der Prädikatenlogik. Somit ergibt sich die algebraische Spezifikation als logische Grundlage der hier betrachteten Spezifikationssprache.

5.2. Grundlagen der Graphtransformation

In diesem Abschnitt wird eine allgemeine Einführung in die algebraische Graphtransformation gegeben, welche in den nachfolgenden Kapiteln genutzt wird, um die in JPL definierten abstrakten Suchmuster in konkrete Suchmuster, welche auf dem Quelltextgraphen gesucht werden können, zu überführen. Die Grundlagen der in dieser Arbeit verwendeten algebraischen Graphtransformation werden in [Roz97] und [EEP+06] definiert.

In Kapitel 5.1 wurde die Methode der Graphtransformation im Überblick eingeführt. Diese Ausführungen werden im Folgenden theoretisch fundiert und um die Konzepte des Graph Matches, der Graphattributierung und der Negative Application Condition ergänzt, da die in den nachfolgenden Abschnitten beschriebenen Graphregeln diese Konzepte nutzen. In dieser Arbeit wird die Codestruktur als gerichteter Graph beschrieben. Die folgenden Definitionen sind zum großen Teil übernommen aus [Mey00].

Ein gerichteter Graph G ist definiert durch: $G = \{G_{Kn}, G_{Ka}, q^G, z^G, lKn^G, lKa^G\}$, wobei G_{Kn} und G_{Ka} Informationsträger sind, während über die Funktionen $q^G : G_{Kn} \longrightarrow G_{Ka}$ und $z^G : G_{Ka} \longrightarrow G_{Kn}$ Quellknoten und Zielknoten definiert werden. Weiterhin existieren die Funktionen $lKn^G : G_{Kn} \longrightarrow Attr_{Kn}$ und $lKa^G : G_{Ka} \longrightarrow Attr_{Ka}$, welche die Attributierung der Knoten und Kanten über ein Alphabet definieren.

Die Graphtransformation basiert auf der regelbasierten Transformation von Graphstrukturen. Eine Regel besteht aus einer linken und einer rechten Regelseite, welche die Transformation spezifizieren. Falls die Graphstruktur, welche in der linken Regelseite definiert wurde, im zu transformierenden Graphen gefunden wird, so wird diese in die durch die rechte Regelseite gegebene Struktur transformiert. In der algebraischen Graphtransformation wird hier von der direkten Ableitung eines Ausgangsgraphen *G* in den abgeleiteten Graphen *H* unter einer gegebenen Regel r gesprochen: *G-r->H*, wobei *r* in den durch die linke Regelseite definierten Match *m* und die durch die Regel definierte Ableitung *a* unterteilt werden kann. *G-m,a->H.*

Folgende Definitionen bilden die formale Grundlage der algebraischen Graphtransformation [Mey00]:

Definition des Graphmorphismus:

Ein Graphmorphismus von Quellgraph *G* in Zielgraph *G'* besteht aus einem Funktionspaar, welches die entsprechenden Knotenmengen und Kantenmengen aufeinander abbildet: $f = (f_{Kn} : G_{Kn} \to G'_{Kn}, f_{Ka} : G_{Ka} \to G'_{Ka})$.

Definition der Grapherzeugung:

Eine Grapherzeugung definiert eine partielle Beziehung zwischen Elementen der linken Regelseite *L* und der rechten Regelseite *R*, welche festgelegt, welche Knoten und Kanten bei Anwendung der Grapherzeugung erhalten bleiben, erstellt werden, oder gelöscht werden müssen. Die Beziehung zwischen L und R ist gegeben durch einen Graphmorphismus.

Definition des Graphmatches:

Ein Graphmatch $m : L \to G$ für eine Grapherzeugung e ist ein Homomorphismus, welcher Knoten und Kanten von L auf G derart abbildet, dass die Graphstruktur und die Labels erhalten bleiben.

Definition der direkten Ableitung:

Eine direkte Ableitung von Quellgraph Q auf Zielgraph Z ergibt sich aus der Anwendung einer Grapherzeugung e an einem Graphmatch m.

Abbildung 28 zeigt, dass die direkte Ableitung als Verbindungsstruktur zwischen Graphen beschrieben werden kann. Die formale Definition erfolgt als *Pushout*, welcher Graphen als Objekte und Grapherzeugungen als Pfeile darstellt. Die Grapherzeugung zwischen L und R entspricht der abgeleiteten Grapherzeugung e*, welche für den aktuellen Graphen festlegt, welche Elemente erhalten, transformiert oder gelöscht werden. Der Match m, welcher aus der Strukturbeschreibung in L resultiert, hat seine Entsprechung in m* welcher aus der in R definierten Zielstruktur resultiert und im abgeleiteten Zielgraphen Z gefunden wird.

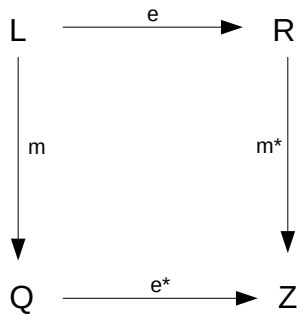


Abbildung 28: Schematische Darstellung der direkten Ableitung als Pushout

Bei der Anwendung von Graphtransformationsschritten, welche auf den vorhergehenden Definitionen basieren, können Situationen entstehen, welche im Rahmen der Konsistenzerhaltung des Graphen problematisch sind. So können ursprünglich einzelne Elemente zu einem Element verschmolzen werden, woraus resultieren kann, dass sowohl der Erhalt, als auch die Löschung eines Elements gleichzeitig spezifiziert werden können. Ein weiteres Problem ist durch „hängende Kanten“ gekennzeichnet. Diese können entstehen, wenn ein Knoten gelöscht wird, ohne gleichzeitig alle mit diesem verbundenen Kanten zu löschen. Daraus resultiert ein inkonsistenter Graph, welcher der in dieser Arbeit verwendeten Definition eines Graphen aus Kapitel 2.1. nicht mehr entspricht.

Um diesen Problemen zu begegnen, kann als Grundlage der zu verwendenden Graphtransformation der Single- oder Double-Pushout-Ansatz eingesetzt werden, welche diese problematischen Situationen ausschließen. Ein ausführlicher Vergleich der Ansätze wird in [EHK+97] gegeben:

Charakterisierung des Single-Pushout-Ansatzes:

Der Single-Pushout-Ansatz (SPA) definiert eine direkte Ableitung entsprechend dem zuvor dargestellten Ansatz, wobei die Löschung eines Elements im Konfliktfall der Erhaltung eines Elements vorgezogen wird. Dies bedeutet für die oben beschriebenen Problemsituationen:

- Wird durch eine direkte Ableitung gleichzeitig der Erhalt und die Löschung eines Elements spezifiziert, so wird das Element gelöscht.
- Falls hängende Kanten im Rahmen einer Ableitung erzeugt werden, so werden diese gelöscht.

Charakterisierung des Double-Pushout-Ansatzes:

Der Double-Pushout-Ansatz (DPA) definiert eine direkte Ableitung entsprechend dem weiter unten dargestellten Ansatz, wobei die Erhaltung eines Elements im Konfliktfall der Löschung eines Elements vorgezogen wird. Dies bedeutet für die oben beschriebenen Problemsituationen:

- Wird durch eine direkte Ableitung gleichzeitig der Erhalt und die Löschung eines Elements spezifiziert, so wird die Transformation nicht durchgeführt.
- Falls hängende Kanten im Rahmen einer Ableitung erzeugt werden würden, so wird die Transformation nicht durchgeführt.

Ableitung im Rahmen des DPA:

Eine direkte Ableitung besteht aus zwei „Klebegraphen“, welche wiederum als Pushouts formalisiert werden. Hierbei wird zwischen der linken Regelseite L und der rechten Regelseite R ein Schnittstellengraph definiert, welcher alle zu löschenden Elemente enthält. Der Kontextgraph K wird hergeleitet aus dem gegebenen Quellgraphen Q, aus dem die Elemente gelöscht werden, welche in S enthalten sind. Diese Operation ist im Pushout 1 definiert. Der Zielgraph Z wird erstellt durch das Hinzufügen aller Elemente, welche in R, aber nicht in K enthalten sind (Pushout 2).

Folgende Bedingungen stellen die Konsistenz des Zielgraphen sicher:

- Die Identifikationsbedingung erfordert, dass ein zu löschendes Element in L nur einmal in L abgebildet ist.
- Die *dangling condition* erfordert, dass in der Spezifikation der Löschung eines Knotens gleichzeitig die Entfernung aller dem Knoten zugeordneten Kante spezifiziert wird.

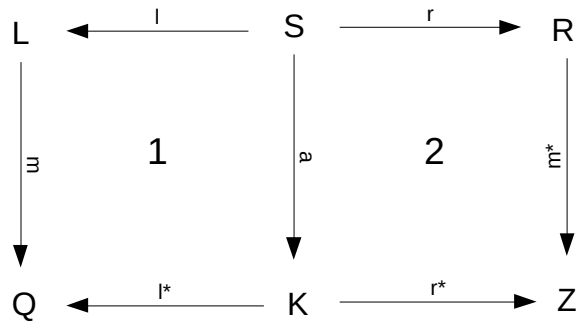


Abbildung 29: Darstellung der direkten Ableitung im DPA als „Klebegraph“

Da durch die Nutzung des DPA ein höheres Maß an Kontrolle über den Zielgraphen gegeben ist als bei Anwendung des SPA, wird im Folgenden der DPA verwendet. Diese Kontrolle erfordert zwar einen höheren Spezifikationsaufwand; da aber in dieser Arbeit der Fokus auf Regeln liegt, welche keine Elemente löschen, kann dieser Aspekt vernachlässigt werden.

Negative Anwendungsbedingungen (Negative Application Conditions):

Eine NAC wird als Vorbedingung zur Ausführung einer Graphregel spezifiziert. Eine NAC ist definiert über eine Graphstruktur. Ist diese Struktur im zu transformierenden Quellgraphen enthalten, so wird die zugehörige Regel nicht ausgeführt. Zu einer Regel können mehrere NACS spezifiziert werden. Negative Anwendungsbedingungen werden formal definiert in [EEP+06] und [HHT96].

Zur Vereinfachung der Regelnotation wird im Folgenden von den unten aufgeführten Prämissen ausgegangen:

- Es ist implizit eine mit der rechten Regelseite identische NAC gegeben. Dies vereinfacht die Regeldarstellung, da diese NAC für alle in dieser Arbeit beschriebenen transformierenden Regeln eine Vorbedingung darstellt.
- Wird als Regeldarstellung nur ein Graph spezifiziert, so werden alle spezifizierten Elemente erhalten und keine weiteren Elemente durch diese Regel erzeugt. Mit diesem Regeltyp werden Graphen nur gesucht und nicht verändert.
- Das Mapping, d.h. die Zuordnung von Elementen der linken und rechten Regelseite, sowie der NACs durch IDs erfolgt in den entsprechenden Knoten. Kanten zwischen zwei zu erhaltenden Knoten werden ebenfalls erhalten.
- Graphregeln werden so häufig wie möglich ausgeführt, soweit im Regelablauf nichts Abweichendes angegeben wurde.

Typgraph:

Der Typgraph erfasst die Vererbungsbeziehungen zwischen den einzelnen Knoten- und Kantentypen.

Für alle Knotentypen KnT gilt: $KnT = \{KnT_{1,} KnT_2 ... KnT_x\}$.

Für alle Kantentypen KaT gilt: $KaT = \{KaT_{1,} KaT_2 ... KaT_x\}$.

Dies bedeutet, dass mehrere Sub-Knoten- und Kantentypen von einem übergeordneten Knoten- und Kantentyp erben können. Der in dieser Arbeit verwendete Typgraph ergibt sich aus den Sortendefinitionen in Kapitel 4.1, welche die verwendeten Typen(mengen) und ihre Relationen darstellt.

Nachdem nun der theoretische Hintergrund der im Folgenden eingesetzten Graphregeln umrissen wurde, stellt das folgende Kapitel die zwei grundlegenden Vorgehensweisen zur Ableitung eines JPL-Musters vor.

5.3. Vorgehensweisen zur Ableitung eines JPL-Musters

In diesem Abschnitt werden die zwei grundlegenden Vorgehensweisen zur Ableitung des JPL-Musters und der Durchführung der Mustersuche allgemein erläutert. Hierzu muss initial das JPL-Muster als JPL-Graphen repräsentiert werden. Abbildung 30 zeigt beispielhaft, wie eine modulare JPL-Spezifikation als JPL-Graph dargestellt wird.

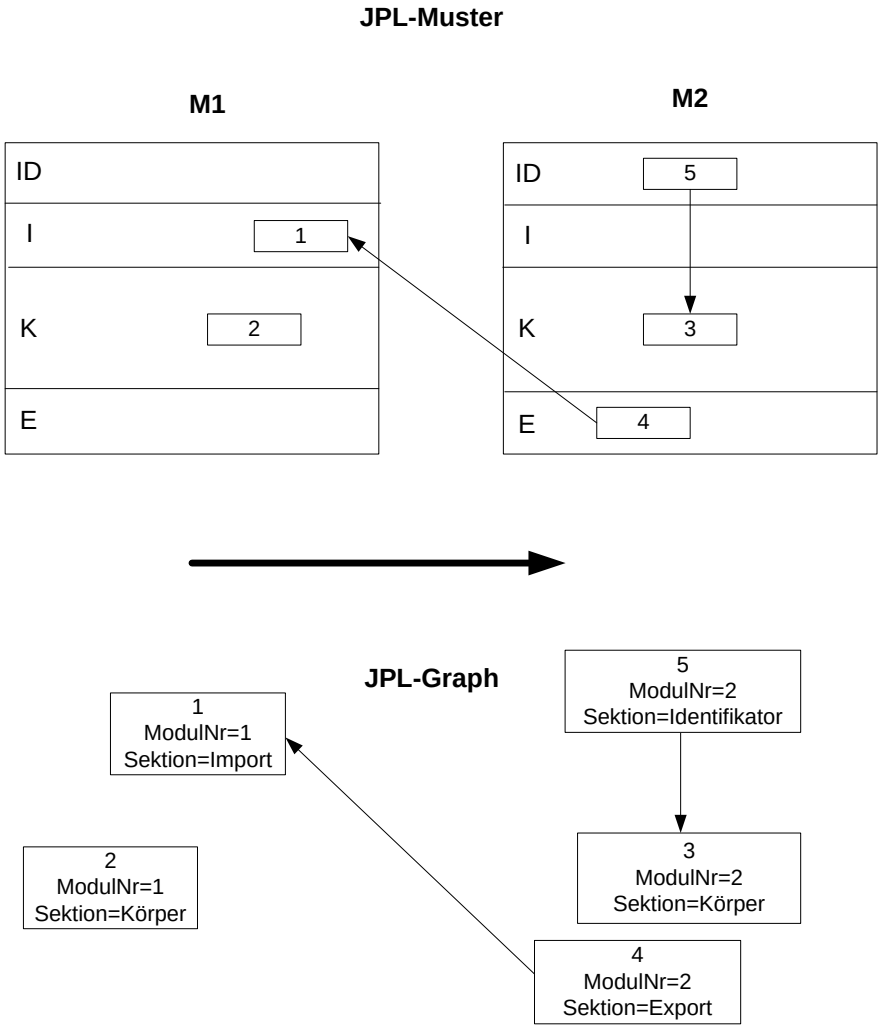


Abbildung 30: Ableitung der modulbasierten JPL-Spezifikation zum JPL-Graphen

Die modulbasierte Darstellung des JPL-Musters wird in den JPL-Graphen abgeleitet, indem die Knoten und Kanten ein zusätzliches Attribut erhalten, das angibt in welcher Sektion diese spezifiziert wurden. *Sektion*: *(Identifikator, Export, Körper, Import).* Weiterhin wird das Attribut *ModulNr* eingeführt, das festlegt, welchem Modul das entsprechende Element zugeordnet ist. Die Module werden hierfür von eins an aufsteigend nummeriert.

Allgemeine initiale Ableitungsschritte:

Initiale Ableitungsschritte (Graphregeln) erzeugen Hilfsstrukturen im JPL-Graphen, die zur Identifikation der Muster in der Quellcoderepräsentation notwendig sind. Hierbei werden die Teilgraphen in der Körper-, Import-, und Exportsektion zu dem in der Identifikatorsektion beschriebenen Gültigkeitsbereich in Beziehung gesetzt.

Quellgraph: JPL-Graph der aus der vom Nutzer eingegebenen JPL-Spezifikation abgeleitet wurde.

Zielgraph: JPL-Graph, der um Zusatzstrukturen angereichert wurde, so dass die implizit durch die Moduldarstellung der JPL-Spezifikation gegebene Semantik durch den JPL-Graphen, bzw. den aus diesem abgeleiteten Suchmustern, realisiert wird.

Ergänzungen:

- Der Graph wird um Knoten ergänzt, die sicherstellen, dass das Muster nur in dem durch die Identifikatorsektion spezifizierten Gültigkeitsbereich gesucht wird.
- Es werden Strukturen generiert, die sicherstellen, dass die in einer Import-Sektion spezifizierten Variablen, im Gültigkeitsbereich, in den sie importiert wurden, auch verwendet werden.
- Der Graph wird um Knoten ergänzt, die sicherstellen, dass im Import spezifizierte Variablen nicht im Gültigkeitsbereich (s. vorheriger Punkt) deklariert werden.

Schritte im Detail:

1. Erstellung der *Bereichsmarkierungsknoten*, die nachfolgend mit den Knoten ihres Moduls verbunden werden. Über diese Knoten werden während der Mustersuche die Gültigkeitsbereiche identifiziert. Die Graphregel ist im Anhang in Abbildung 97 aufgeführt.
2. Die Knoten im Körper und Export, werden mit Knoten zur Bereichsmarkierung für die einzelnen Module verbunden. Alle JPL-Knoten werden mit den entsprechenden Knoten zur Bereichsmarkierung verbunden, wobei die Kanten mit einem Attribut versehen werden das angibt, ob der Schnittstellenknoten im Import oder Export spezifiziert wurde. Dies stellt sicher, dass die Strukturen nur in Bereichen gefunden werden, welche zuvor durch die Identifikatorsektion ermittelt wurden. Für Ableitungsvarianten der indirekten Variablenübergabe kann diese Attributierung entfallen. Die Graphregeln sind im Anhang in den Abbildung 99 und Abbildung 100 dargestellt.
3. Die JCG- und AJSDG-Elemente im Import der einzelnen Module werden mit einem Knoten vom Typ *MarkAll* verbunden der anzeigt, dass diese Elemente nicht im durch die Identifikatorsektion vorgegebenen Bereich gesucht werden dürfen. Dies ist notwendig, um im Laufe der Mustersuche Knoten identifizieren zu können, die in keinem Gültigkeitsbereich liegen. Die Graphregeln sind im Anhang in Abbildung 102 und Abbildung 103 beschrieben. Hinweis: Die Köpfe eines Bereichs (Methodenkopf, Kopf einer Bedingung) und deren Parameter sind nicht Teil des Bereichs, der durch ein Modul spezifiziert wird.

Das oben beschriebene Vorgehen wurde gewählt, da das Ziel verfolgt wird, die aus dem JPL-Graphen abgeleiteten Graphen der Implementierungsvarianten direkt über Transformationsregeln erstellen zu können. Alternativ könnten die Knoten, die nicht in einem Bereich identifiziert werden dürfen, über NACs ermittelt werden. Hier würde der Schritt der umfangreichen Markierung des Quelltextgraphen entfallen, allerdings müssten nun zusätzliche NACs im Suchablauf in allen Implementierungsvarianten betrachtet werden.

4. Die Ergebnisknoten werden festgelegt, d.h. es wird vorgegeben, welche Information bei einem gefundenen Muster an den Nutzer übergeben werden soll.

Alternativen zur Ableitung des JPL Graphen:

Zur Ableitung des JPL-Graphen in Strukturen, welche die verschiedenen Implementierungsvarianten der Variablenübergabe abdecken, existieren zwei grundlegende Optionen:

1) Die zu suchenden Mustervarianten werden als Gesamtgraph erstellt, d.h. pro Variante wird eine Struktur generiert, die alle Elemente aus allen Sektionen aller im Muster spezifizierten Module, sowie die jeweiligen Variablenübergabevarianten, enthält.

   - Vorteile:
     - Der Algorithmus zur Mustersuche ist im Vergleich zu der im Folgenden betrachteten Variante einfacher, da ein vollständiges Muster in einem einzigen Schritt gesucht werden kann, ohne dass weitere Prozesschritte zur Analyse der gefundenen Matches notwendig sind.
     - Sobald ein Muster gefunden wurde kann der Suchvorgang beendet werden. Es müssen keine weiteren Prozessschritte für den Ablauf der Mustersuche durchgeführt werden, da im Match die zu suchende Implementierungsvariante vollständig enthalten ist und somit erkannt wurde.
   - Nachteile:
     - Initial ist ein hoher Vorbereitungsaufwand zur Erstellung der Varianten notwendig, da für jede Variablenübergabevariante ein neues Gesamtmuster generiert werden muss. Die Anzahl der Mustervarianten sollte vom Nutzer bei Wahl dieser Alternative eingeschränkt werden, indem er z.B. nur nach einigen zuvor festgelegten Variableübergabestrukturen sucht.
     - Der Suchvorgang nach einem umfangreichen komplexen Muster ist langsamer als die Suche nach kleineren Mustern.
     - Pro Modul darf nur ein JPL-Schnittstellenknoten verwendet werden. Falls in einem Muster, das aus mehreren Modulen besteht, mehrere JPL-Schnittstellenknoten spezifiziert werden, so müssen sich diese auf unterschiedliche Variablen beziehen und die Zuordnung der Variablen zu Klassen kann nicht betrachtet werden (Problem doppelter Strukturen in abgeleiteten Suchmustern).

- Anwendungskriterien:
    - Diese Vorgehensweise ist anzuwenden, wenn die Spezifikation in der Identifikatorsektion sehr generisch gehalten und somit eine große Anzahl potentieller Matches pro Modul möglich ist.
    - Diese Vorgehensweise ist anzuwenden, wenn das Suchmuster Strukturen enthält, die häufig im Quelltext vorkommen, so dass eine hohe Anzahl an Matches für die Elemente in der Körpersektion wahrscheinlich ist.
    - Es dürfen nur wenige Variablenübergabestrukturen im JPL-Muster spezifiziert sein, da ansonsten die Anzahl der zu erstellenden Muster zu hoch wird. Alternativ muss die Zahl der zu suchenden Übergabevarianten vom Nutzer eingeschränkt werden, indem z.B. nur direkte oder nur indirekte Variablenübergaben betrachtet werden.

2) In der zweiten Variante wird das JPL-Muster in seine Module und deren Sektionen aufgeteilt betrachtet und pro Sektion werden einzelne Suchvorgänge durchgeführt. Während des Ablaufs der Mustersuche werden die Matches der einzelnen Module zur Identifikatorsektion, Körpersektion und zu den Übergabevarianten der Import- und Exportsektionen miteinander kombiniert, so dass ein gültiger Gesamtmatch gefunden wird. Während in der ersten Variante der Schwerpunkt auf der Ableitung und nachfolgend der Suche von vollständigen Suchmustern lag, steht nun die Unterteilung der Muster und die Kombination der gefundenen Teilmuster zur Gesamtmusteridentifikation im Vordergrund. Hieraus ergeben sich folgende Vorteile und Nachteile:

    - Vorteile:
        - Kleine Muster werden schneller in Graphen gefunden als große Strukturen.
        - Pro Modul dürfen mehrere JPL-Schnittstellenknoten verwendet werden.
    - Nachteile:
        - Komplexer Ablauf der Suche, da für die Identifikation eines Gesamtmatches mehrere Matches kleinerer Einzelmuster kombiniert betrachtet werden müssen.
        - Bei einer großen Zahl von Matches der Teilmuster steigt die Dauer der Mustersuche erheblich an.
        - Falls in einer hierarchischen Struktur in den Körpersektionen des übergeordneten und untergeordneten Moduls die gleichen Elementtypen beschrieben werden, so kann beim Ablauf der Mustersuche nicht ausgeschlossen werden, dass diese Elemente des Quellcodegraphen mehrfach als valider Match erkannt werden.

- Anwendungskriterien:
  - Eine möglichst detaillierte Spezifikation in der Identifikatorsektion der einzelnen Module unterstützt den Suchprozess, da diese den Suchraum für die Teilmuster entsprechend einschränkt.
  - Die Strukturen in den Modulkörpern sollten nicht zu allgemein formuliert werden (möglichst keine generischen Strukturen, wie z.B. einen *if*-Knoten, ohne weitere Beziehungen), da sonst eine große Anzahl an Matches der Teilmuster untersucht und kombiniert werden müssen.
  - Diese Variante ist anzuwenden bei Mustern, in denen viele Übergaben spezifiziert sind und die Art der Übergaben möglichste offen gelassen werden soll.

Die Erstellung von Suchmustern zu Variante zwei kann über unterschiedliche Arten erfolgen. Zum einen können die sektionsbasierten Teilmuster aus den vollständig abgeleiteten Gesamtgraphen übernommen werden, oder zum anderen können die Übergabestrukturen aus einem Katalog entnommen werden. Die Übernahme aus einem Katalog vereinfacht die Mustererstellung verglichen mit den weiteren Varianten, allerdings werden mögliche Abhängigkeiten zwischen Strukturen im Körper der Module und den Übergabestrukturen nicht betrachtet. Dies bedeutet, dass über dieses Verfahren nur Muster gesucht werden dürfen, die keine Verbindungen zwischen den Strukturen der Körpersektionen und den Schnittstellenstrukturen besitzen.

Generell ergibt sich die Vielfalt der abgeleiteten Suchmuster zu einer JPL-Spezifikation aus den unterschiedlichen zu erkennenden Implementierungsvarianten, welche aus der Verwendung von JPL-Schnittstellenknoten/-kanten resultieren. Im Folgenden werden Graphtransformationsregeln beschrieben, welche die JPL-Schnittstellenstrukturen in JCG-Variableübergabestrukturvarianten ableiten, von denen im JPL-Muster abstrahiert wurde. Abhängigkeiten zu Elementen in Körpersektionen werden hierbei berücksichtigt.

5.4. Ableitung der JPL-Schnittstellenstrukturen

Die in der JPL-Syntax enthaltenen JPL-Schnittstellenknoten und JPL-Schnittstellenkanten kapseln JCG-Strukturen, welche die Übergabe von primitiven Datentypen oder Objekttypen zwischen Gültigkeitsbereichen repräsentieren. In diesem Abschnitt wird die Ableitung der abstrakten JPL-Schnittstellenknoten und –kanten in konkrete JCG-Graphstrukturen beschrieben. Hierbei wird für jede Übergabestruktur eine Regelmenge erstellt. Die Anwendung dieser Regeln auf ein gegebenes JPL-Suchmuster generiert die JCG-Suchmuster (Implementierungsvarianten) zu dieser Spezifikation, die nachfolgend im Rahmen der Ausführung der Mustersuche (s. Kapitel 5.9. und 5.10) angewendet werden.

Formal

Die Funktion $f : JPLSK(Export) - SKa - JPLSK(\mathrm{Im}\, port) \rightarrow \sum IEG_{JCG}$, welche in Kapitel 4.1. benannt wurde, wird im Folgenden durch Graphtransformationsregeln realisiert, d.h. Strukturen aus JPL-Schnittstellenknoten und -kanten werden in JCG-Strukturen abgeleitet.

Allgemeine Darstellung der Ableitungsregeln zu JPL-Schnittstellenstrukturen

Die im Weiteren dargestellten Regeln zur Erzeugung der Suchmustervarianten zu JPL-Schnittstellenstrukturen können allgemein durch die unten aufgeführte Regel beschrieben werden. Die Schnittstellenknoten der zueinander in Beziehung stehenden Module M1 und M2 werden in eine Suchmustervariante abgeleitet, die ausschließlich JCG-Elemente enthält.

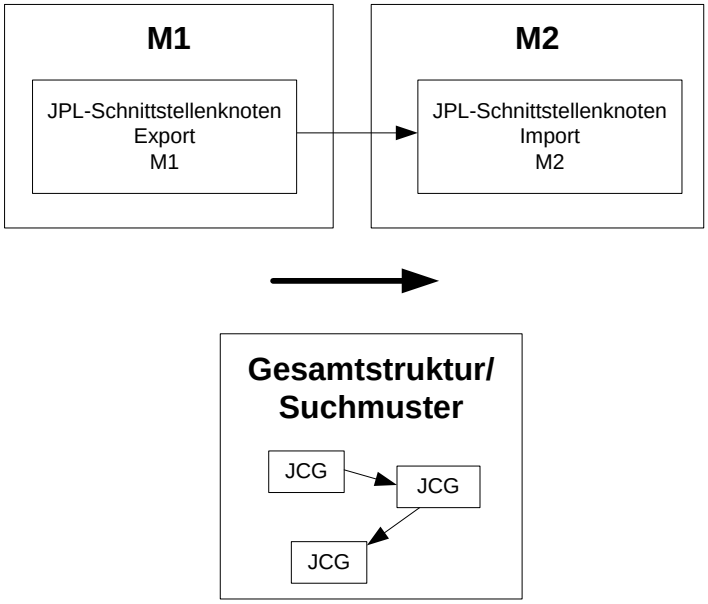


Abbildung 31: Allgemeine Regel zur Ableitung der JPL-Schnittstellenknoten

Die Regeln zur Erstellung der Suchmuster zu den verschiedenen Implementierungsvarianten werden als Menge lokaler Graphtransformationsregeln beschrieben. Die Regeln setzen auf dem JPL-Graphen auf.

Grundlegende Alternativen der Ableitung komplexer Muster und ihr Einfluss auf die Auflösung der JPL-Schnittstellenstrukturen

Wie im vorherigen Kapitel erläutert existieren zur Zerlegung eines Musters in die verschiedenen Mustervarianten zwei grundlegende Optionen:

1) Zu einem gegebenen JPL-Muster werden vollständig ausspezifizierte Muster erstellt, welche die Elemente aller Module und aller Sektionen enthalten.
2) Die Strukturen in den verschiedenen Sektionen werden im Rahmen der Musterableitung separat betrachtet und nachfolgend in einem Regelablauf kombiniert.

Eine ausführliche Betrachtung wurde in Kapitel 5.3. gegeben. Im Folgenden wird die Auswirkung dieser Varianten auf die Ableitung der JPL-Schnittstellenstrukturen dargestellt:

Ableitung für Variante eins: Die Schnittstellenstrukturen werden zusammen mit dem vollständigen JPL-Graphen betrachtet und die Transformationen auf diesem durchgeführt, um die zu suchenden Mustervarianten zu erhalten.

Ableitung für Variante zwei: Die JPL-Schnittstellenstrukturen werden separat vom Gesamtmuster betrachtet, so dass die im Folgenden beschriebenen JCG-Übergabestrukturen aus einem Katalog entnommen werden können. Falls Verbindungen zu weiteren Elementen der Körpersektion bestehen, müssen diese in die Schnittstellestruktur übernommen werden (zu Details sh. Kapitel 5.7.).

Die beiden folgenden Kapitel stellen die für Alternative eins notwendigen Transformationsschritte dar. Falls die Ableitung im Rahmen von Alternative zwei durchgeführt werden soll, so müssen die JPL-Schnittstellenstrukturen entsprechend des in Kapitel 5.7. gegebenen Prozesses abgeleitet werden. Dort wird auf die unten gezeigten Übergabevarianten zurückgegriffen.

Die weitere Strukturierung des Kapitels orientiert sich an den unterschiedlichen Gültigkeitsbereichen der Sprache Java und den verschiedenen Möglichkeiten, Variablen zwischen diesen zu übergeben. Folgende Gültigkeitsbereiche werden unterschieden:

- **Interface:** Ein Interface gibt Methoden vor, welche von der Klasse, die dieses implementiert, realisiert werden müssen. Weiterhin werden durch Interfaces Konstanten vorgegeben, welche von den implementierenden Klassen genutzt werden können.
- **Abstrakte Klasse:** Abstrakte Klassen geben Deklarationen von Membervariablen, Klassenvariablen und Methodenrümpfe vor, welche nachfolgend von der realisierenden Klasse implementiert werden müssen.
- **Klasse:** Eine Klasse enthält Methoden, Konstruktoren, Klassenvariablen und Membervariablen, welche von anderen Klassen aus referenziert werden können.

Die oben aufgeführten Bereiche können im Suchmuster über die Modulverbindungskanten und JCG-Schnittstellenelemente gekoppelt werden, welche in Kapitel 4.5., Methode 1 beschrieben wurden. Die Ableitung der Übergabevarianten aus den JPL-Schnittstellenstrukturen wird in den unten beschriebenen *direkten Aufrufen* dargestellt.

Abgrenzung:

- **Innere Klasse:** Die innere Klasse ist innerhalb einer äußeren Klasse eingebettet und kann nur zusammen mit dieser instanziiert werden. Sie hat Zugriff auf alle Variablen der äußeren Klasse.
- **Statische innere Klasse:** Die Klasse kann als unabhängig von der diese einbettenden äußeren Klasse betrachtet werden und kann nur auf deren Static-Variablen zugreifen.

Da die Struktur der inneren Klasse identisch ist mit der Struktur einer äußeren Klasse und die Struktur eines Zugriffs auf äußere Variablen strukturell identisch ist mit dem

Zugriff einer Methode innerhalb der äußeren Klasse, wird diese in den folgenden Übergabestrukturen nicht explizit aufgeführt. Die Identifizierung einer Übergabe in eine innere Klasse erfolgt analog zur Identifizierung einer Übergabe in eine Methode. Da das Verhalten der statischen inneren Klasse sich nur durch den dynamische Ablauf und der hieraus resultierenden Zustände (als Objekt instanziiert oder nicht) von einer inneren Klasse unterscheidet, wird diese im Folgenden ebenfalls nicht explizit betrachtet.

- **Methode:** Einer Methode können im Rahmen des Methodenaufrufs Parameter übergeben werden. Weiterhin kann im Gültigkeitsbereich der Methode direkt auf Membervariablen der einbettenden Klasse zugegriffen werden. Über die *return*-Anweisung können Variablenwerte aus dem Gültigkeitsbereich heraus übergeben werden.
- **Konstruktor:** Einem Konstruktor werden Parameter im Rahmen der Objektinstanziierung übergeben. Weiterhin kann im Gültigkeitsbereich des Konstruktors direkt auf Membervariablen der einbettenden Klasse zugegriffen werden.

Die oben aufgeführten Bereiche können explizit über die Modulverbindungskanten und JCG-Schnittstellenelemente gekoppelt werden, welche in Kapitel 4.5. beschrieben wurden. Die Ableitung der Übergabevarianten aus den JPL-Schnittstellenstrukturen wird über die unten beschriebenen *indirekten Aufrufe* dargestellt.

- **Bedingungen, Schleifen:** Bedingungsparameter in Schleifen oder *if*-Bedingungen können direkt auf die Variablen der sie einbettenden Strukturen, wie z.B. eine Methode und Klasse zugreifen. Sowohl Bedingung als auch Schleifen definieren einen eigenen Gültigkeitsbereich.
- **Try-catch-Blöcke:** Try-catch Blöcke sind, wie auch if-then-else-Bedingungen, gekoppelte Strukturen, wobei an den Catch-Block Parameter übergeben werden können.

Die oben aufgeführten Bereiche können explizit über die Modulverbindungskanten und JCG-Schnittstellenelemente gekoppelt werden, welche in Kapitel 4.7. gezeigt wurden. Die Ableitung der Variablenübergabestrukturen aus den JPL-Schnittstellenstrukturen wird über die unten beschriebenen *direkten Aufrufe* dargestellt.

<u>Folgende Variablentypen werden unterschieden (bezogen auf ihren Gültigkeitsbereich):</u>

- Membervariable: Variable einer Klasse
- Lokale Variable: Die Variable ist in einem Gültigkeitsbereich deklariert, und kann damit in diesem und in ihn eingebetteten Bereichen verwendet werden.
- Parameter einer Methode: Variablenwerte werden im Rahmen des „call by value“ an die Parameter einer Methode übergeben.
- Parameter eines Return Statements: Der Wert des Rückgabeparameters einer Methode wird an den aufrufenden Ausdruck übergeben.

Abgrenzung:

Die Repräsentation statischer Variablen ist in der JCG identisch mit dem Knoten einer Membervariablen oder lokalen Variablen. Allerdings ist das Attribut *static* im Variablenknoten auf *true* gesetzt. Da der Zugriff auf statische Variablen analog zu den vorgenannten spezifiziert wird, werden diese im JPL-Muster strukturell nicht unterschieden. Konstanten unterscheiden sich ebenfalls nicht in ihrer strukturellen Darstellung, sondern lediglich in der Belegung des Knotenattributs *final* auf *true*. Somit wird auch dieser Typ im Weiteren nicht explizit betrachtet.

Folgende Status von Variablen werden unterschieden:

- Public Klassen-/Membervariable: Auf die Variable kann von Gültigkeitsbereichen außerhalb der Klasse zugegriffen werden.
- Protected Klassen-/Membervariable: Nur Klassen die von der Klasse, welche die Variable enthält, erben, können direkt auf diese zugreifen.
- Private Klassen-/Membervariable: Die Membervariable darf nur innerhalb des Gültigkeitsbereichs der Klasse verwendet werden.

Diese Status werden im Folgenden nur explizit betrachtet, wenn sie für die Regelausführung, bzw. die entsprechende Implementierungsvariante notwendig sind, oder im Suchergebnis dargestellt werden sollen.

Ableitung von in JPL-Schnittstellenstrukturen gekapselten JCG-Strukturen

Im diesem Abschnitt werden die Übergabestrukturen dargestellt, welche sich aus den zuvor beschriebenen Gültigkeitsbereichen und Variablentypen ergeben. Es wird zwischen direkten und indirekten Variablenzugriffen unterschieden. Während für den direkten Aufruf die Variable direkt referenziert wird, d.h. es wird auf eine Variable zugegriffen, die in einem hierarchisch höherstehenden Gültigkeitsbereich deklariert wurde, wird die Variable im indirekten Fall über Hilfskonstrukte übertragen.

Direkte Aufrufe:

- Aufruf einer Membervariablen.
- Aufruf einer vererbten Membervariablen.
- Aufruf einer Membervariablen einer Klasse, die eine abstrakte Klasse realisiert.
- Aufruf einer Membervariable aus einer externen Klasse.
- Aufruf der Konstante eines Interfaces.
- Aufruf lokaler Variablen von eingebetteten Gültigkeitsbereichen.

Indirekte Aufrufe:

- Eine Variable wird als Parameter an eine Methode übergeben.
- Eine Variable wird über die Return-Anweisung zurückgegeben.
- Eine Variable wird über eine Getter- oder Setter-Methode aufgerufen.

Im folgenden Kapitel werden die Regeln zur Ableitung von JPL-Schnittstellenstrukturen für Variablenübergaben durch direkte Aufrufe erläutert. Regeln zu Aufrufen, welche über identische JCG-Graphstrukturen dargestellt werden, sind entsprechend zusammengefasst.

5.5. Varianten für die Variablenübergabe durch direkte Aufrufe

In diesem Kapitel wird die Ableitung der JPL-Schnittstellenstrukturen für die Implementierungsvarianten der direkten Aufrufe über Graphtransformationsregeln auf dem JPL-Graphen beschrieben.

Zur Erinnerung: Es ist möglich Knoten vom Typ *accessToReferenceTypeVariable* und *accessToPrimitiveTypeVariable* aus dem Körper eines Moduls mit einem JPL-Schnittstellenknoten im Import oder Export eines Moduls zu verbinden. Diese Kanten müssen im Verlauf des Ableitungsprozesses erhalten bleiben, so dass die Auflösung der Schnittstellenknoten die Generierung eines Hilfknotens des Typs *JPLObjectDeclaration* erfordert, welcher diese Kanten übernehmen kann.

Über die Regeln in der folgenden Abbildung wird dieser Ableitungsschritt realisiert. Durch die erste Regel wird ein *JPLObjectDeclaration*-Knoten erstellt, welchem nachfolgend alle Verbindungen der *JPLObject*-Knoten zugewiesen werden. Durch die zweite Regel wird die JPL Beziehung markiert, welche aktuell betrachtet wird. Beide Regeln werden jeweils nur einmal ausgeführt. Durch die dritte Regel werden alle eingehenden Kanten zu den *JPLObject*-Knoten auf den *JPLObjectDeclaration*-Knoten umgeleitet. Knoten vom Typ *JPLObject* sind hier ausgenommen.

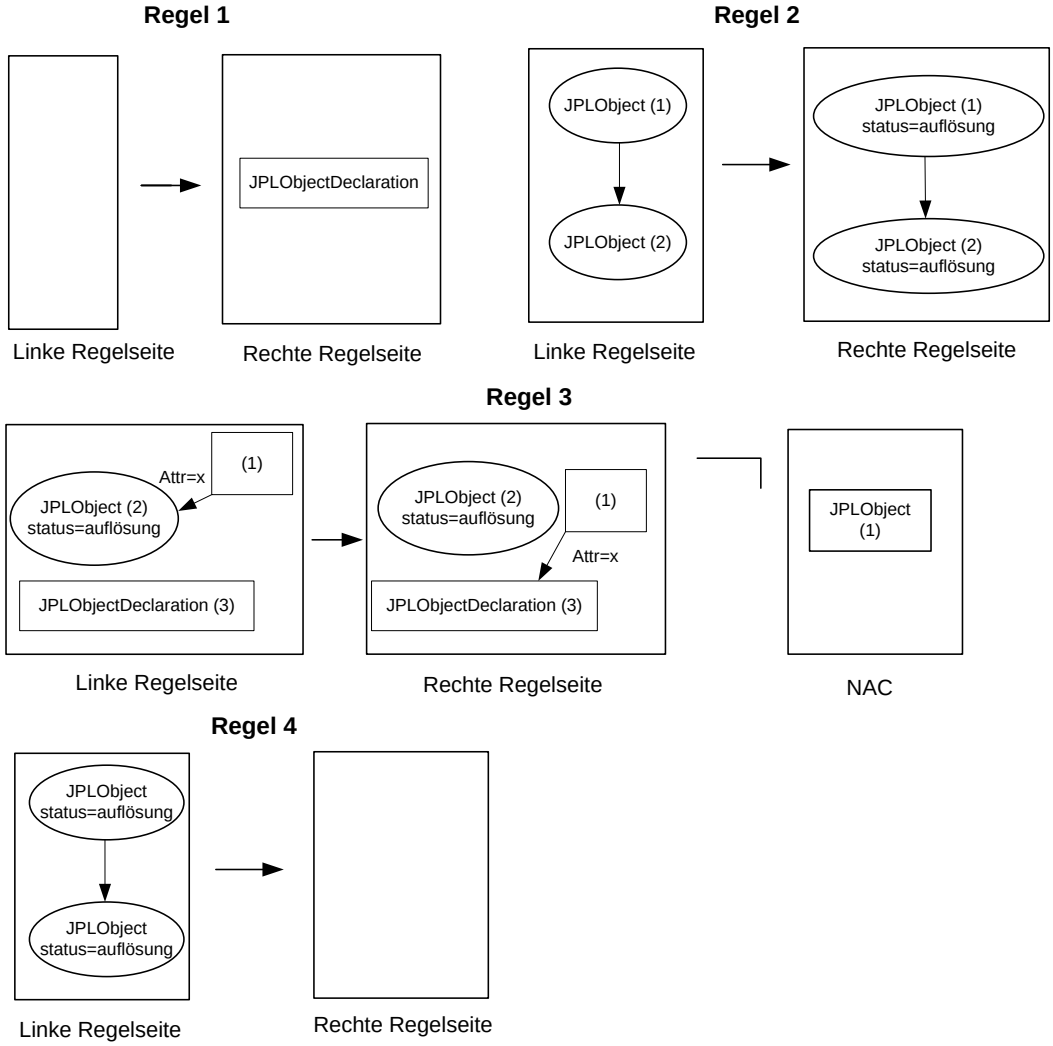


Abbildung 32: Regelmenge zur Auflösung der JPL-Schnittstellenknoten

Nachfolgend werden die *JPLObject*-Knoten mit dem Status a*uflösung* gelöscht. Durch diese Verschmelzung der Struktur zweier miteinander verbundener *JPLObject*-Knoten, die jeweils im Im- und Export zweier Module spezifiziert wurden, zu einem Knoten vom Typ *JPLObjectDeclaration*, ist die allgemeine Graphstruktur des Musters für den direkten Variablenaufruf bereits erstellt.

Bedingungen des Regelsatzes:

Vorbedingung: Zwei durch eine Kante verbundene Knoten vom Typ *JPLObjec*t wurden im Graph spezifiziert.

Nachbedingungen: Ein Knoten vom Typ *JPLObjectDeclaration* wurde erstellt. Kanten, die auf die Knoten vom Typ *JPLObject* gerichtet waren, wurden gelöscht. Kanten, die von den jeweiligen Quellknoten auf den neu erstellten Knoten vom Typ *JPLObjectDeclaration* gerichtet sind, ersetzen die gelöschten Kanten. Die Attributwerte blieben während der Ersetzung erhalten. Die zwei Knoten vom Typ *JPLObject* wurden gelöscht.

Im Folgenden werden die Erweiterungen beschrieben, welche die Art der Übergabe näher analysieren. Das Ziel der Analyse ist es zu ermitteln, aus welchem Quell-Gültigkeitsbereich heraus und in welchen Ziel-Gültigkeitsbereich hinein die Daten übertragen werden. Hierzu werden ausgehend von dem Muster, welches durch die oben beschriebenen Transformationen gebildet wurde, mehrere alternative Mustervarianten erzeugt, um die verschiedenen Übergabevarianten abfragen zu können.

Übergabetyp: Direkter Aufruf einer Membervariable

Folgende Regeln ergänzen den JPL-Graphen um Strukturen, durch die verschiedene Implementierungsvarianten erstellt werden, über welche ermittelt wird, in welchem Kontext eine Deklaration erfolgt. So kann es sich um eine Deklaration in einer Vererbungsstruktur, einem Interface, einer abstrakten Klasse oder um eine einfache Membervariable handeln. Somit kann auch bei sehr abstrakten oder fehlenden Gültigkeitsbereichen der Module, die die Schnittstellenstruktur enthalten, festgestellt werden, aus welcher Struktur heraus die Deklaration erfolgt.

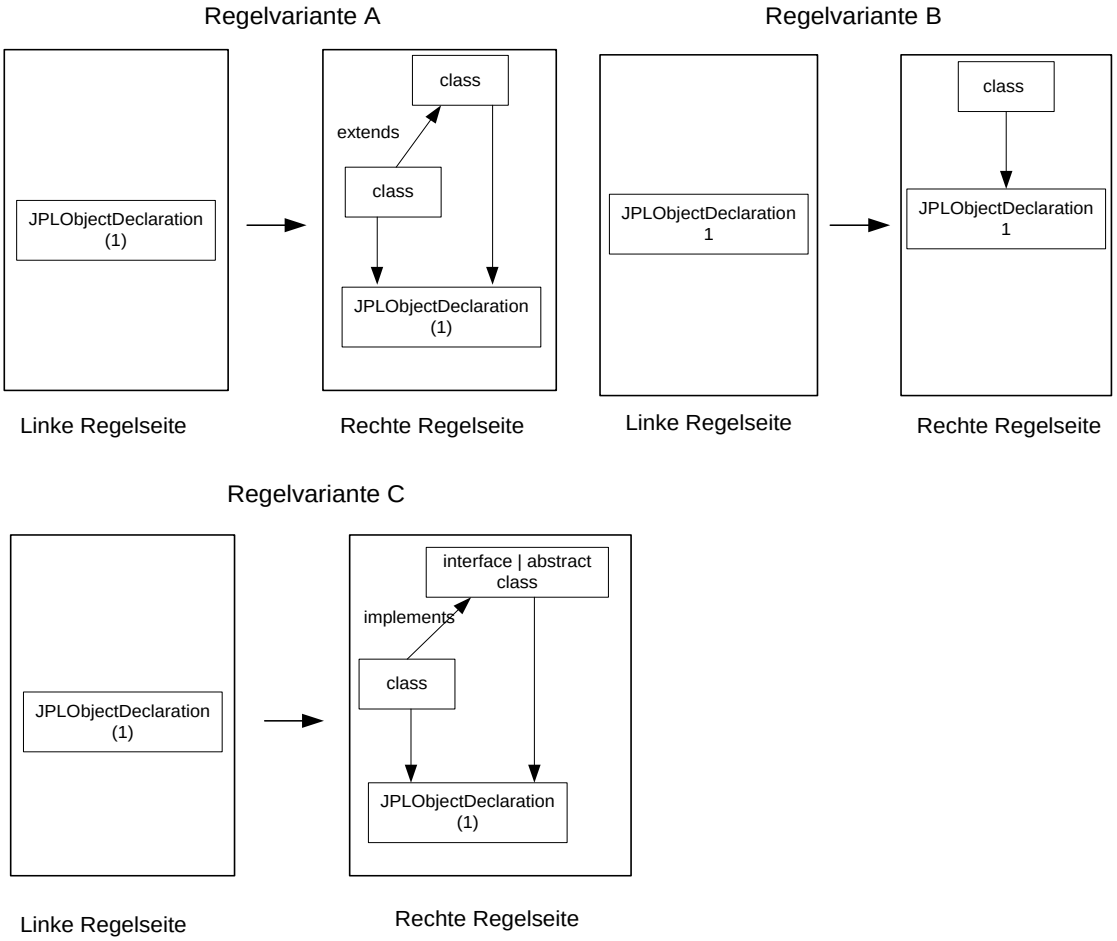


Abbildung 33: Regelvarianten zur Analyse des exportierenden Gültigkeitsbereichs einer Membervariablen

Über Regel A wird auf dem JPL-Graphen die Struktur einer Implementierungsvariante erzeugt, die überprüft, ob es sich um eine Vererbungsstruktur handelt. Durch die Struktur welche Regel B erzeugt wird generell geprüft, ob die Deklaration einer Membervariablen zuzuordnen ist. Durch Regel C wird eine Variante generiert, die den Zugriff auf ein Interface, oder auf eine in einer abstrakten Klasse definierten Variable repräsentiert.

Bedingungen des Regelsatzes:

Vorbedingung: Es existiert ein Knoten vom Typ *JPLObjectDeclaration* im JPL-Graphen.

Nachbedingungen (alternative Strukturen zu Implementierungsvarianten):

Regel A: Der Knoten wurde um eine Vererbungsstruktur ergänzt.

Regel B: Der Knoten wurde um die Struktur einer Membervariablen ergänzt.

Regel C: Der Knoten wurde um eine Struktur zur Verwendung eines Interfaces bzw. einer abstrakten Klasse ergänzt.

Im Folgenden wird der Gültigkeitsbereich identifiziert, aus dem die Variable aufgerufen wird. Initial werden klasseninterne Aufrufe betrachtet. Die folgenden Transformationen auf dem JPL-Graphen erzeugen Implementierungsvarianten, die die verschiedenen möglichen Gültigkeitsbereiche, aus denen die Variablennutzung erfolgt, identifizieren. Somit kann auch bei abstrakten Angaben in der Identifikatorsektion die Art des aufrufenden Gültigkeitsbereichs ermittelt und dem Übungsleiter nachfolgend über Ergebnisknoten mitgeteilt werden:

Über die folgende NAC wird analysiert, ob die über Regel A, B oder C gefundene Membervariable extern referenziert wird. Diese Bedingung muss entsprechend den oben erzeugten Graphen zu den verschiedenen Implementierungsvarianten hinzugefügt werden. Es wird nur der Zugriff auf das Objekt betrachtet. Wenn analysiert werden soll, wo das Objekt instanziiert wird, muss in den folgenden Regeln der Knotentyp *accessToReferenceTypeVariable* durch den Typ *instantiate* ersetzt werden.

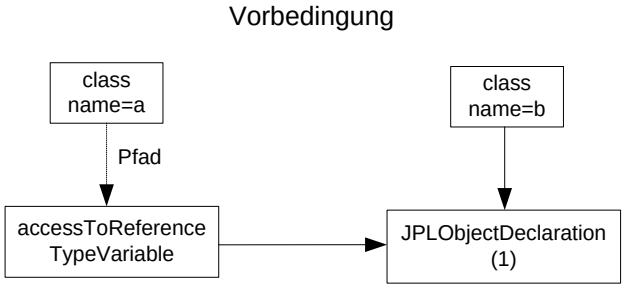


Abbildung 34: NAC zur Gültigkeitsbereichsüberprüfung

Durch die oben dargestellte Vorbedingung wird sichergestellt, dass der Zugriff auf die Membervariable intern erfolgt, da ein Zugriff von einer externen Klasse aus hier vollständig erfasst wird. Zur Erinnerung: Ein Pfad bezieht sich immer auf einen Gültigkeitsbereich, d.h. in diesem Fall den Gültigkeitsbereich einer externen Klasse. Der Knotentyp *accessToReferenceTypeVariable* repräsentiert einen Objektzugriff. Äquivalent kann diese NAC für den Zugriff auf die weiteren JPL-Knotentypen formuliert werden.

Um analysieren zu können, von welchem Gültigkeitsbereich aus der Zugriff auf die Membervariable erfolgt, werden über die folgenden Regeln die entsprechenden Muster zu den verschiedenen Implementierungsvarianten erstellt.

Direkte Übergabe in interne Gültigkeitsbereiche

Hinweis: Die weiteren Graphtransformationen beziehen sich nur auf Objektstrukturen, da die Knotentypen *objectDeclaration* und *accessToReferenceTypeVariable* verwendet werden. Äquivalent können die Regeln für *primitiveDeclaration* und *accessToPrimitiveTypeDeclaration* formuliert werden.

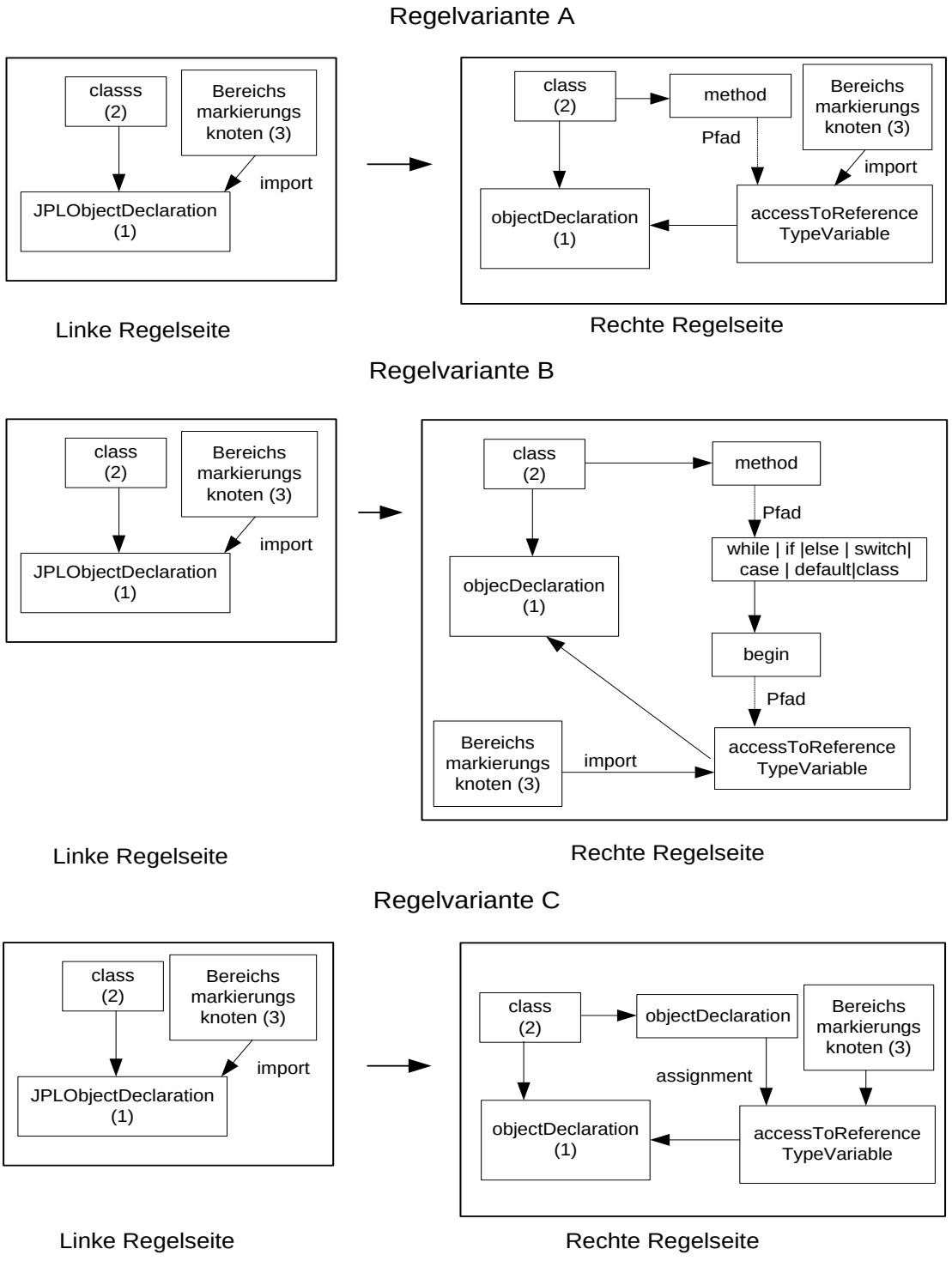


Abbildung 35: Regelvarianten zur Ermittlung des Gültigkeitsbereichs eines Variablenaufrufs

Durch Ausführung der Regel in Variante A wird ein Suchmuster erstellt, dessen Referenz auf die Membervariable aus einer Methode heraus erfolgt. In Variante B wird ermittelt, ob der Aufruf aus einer Bedingung erfolgt, oder z.B. aus einer internen Klasse. Die Knotentypen sind hier bezogen auf die möglichen Gültigkeitsbereiche, die in eine Methode eingebettet werden können. Durch Ausführung der Regel in Variante C wird ein Suchmuster erstellt, das eine Referenzierung im Rahmen der Deklaration einer weiteren Membervariable der Klasse vorsieht. Für Referenzen aus einem Konstruktor muss in den obigen Regeln der Knotentyp *method* durch den Typ *constructor* ersetzt werden. Falls durch einen Match der Variante B eine innere Klasse ermittelt wird, so können im nächsten Schritt auch hier wieder die oben beschriebenen Varianten eingesetzt werden, wobei der *class*-Knoten als innere Klasse attributiert werden muss.

Bedingungen des Regelsatzes:

Vorbedingung: Es existiert ein Knoten vom Typ *JPLObjectDeclaration* im JPL-Graphen. Es ist kein Knoten vom Typ *accessToReferenceTypeVariable* mit diesem verbunden. Mit der *JPLObjectDeclaration* ist der *Bereichsmarkierungsknoten* des importierenden Moduls verbunden.

Nachbedingungen:

Regel Variante A: Der Knoten wurde um eine Struktur ergänzt, die einen Zugriff aus einer Methode repräsentiert. Der Typ *JPLObjectDeclaration* wurde in *objectDeclaration* geändert. Der *Bereichsmarkierungsknoten* wurde auf das referenzierende Element umgehängt. Die Erstellung des Musters ist damit abgeschlossen.

Regel Variante B: Der Knoten wurde um eine Struktur ergänzt, die einen Zugriff aus einem Bereich, der in eine Methode eingebettet ist, repräsentiert. Der Typ *JPLObjectDeclaration* wurde in *objectDeclaration* geändert. Der *Bereichsmarkierungsknoten* wurde auf das referenzierende Element umgehängt. Die Erstellung des Musters ist damit abgeschlossen.

Regel Variante C: Der Knoten wurde um eine Struktur ergänzt, die einen Zugriff von einer weiteren Membervariablen der Klasse repräsentiert. Der Typ *JPLObjectDeclaration* wurde in *objectDeclaration* geändert. Der *Bereichsmarkierungsknoten* wurde auf das referenzierende Element umgehängt. Die Erstellung des Musters ist damit abgeschlossen.

Durch die vorhergehenden Regeln wurden Muster zu Implementierungsvarianten erstellt, welche die interne direkte Referenzierung einer Membervariablen überprüfen. Es ergibt sich folgende Anzahl strukturell unterschiedlicher Muster zu den verschiedenen Implementierungsvarianten:

3 Membervariablenkontexte * 3 Varianten zu internen Zugriffsbereichen = 9 Graphen zu strukturell unterschiedlichen Implementierungsvarianten, die aus einer Übergabestruktur eines JPL-Graphen erzeugt werden.

Regelsatz für Implementierungvarianten bei bestehender Verbindung des JPL-Schnittstellenknotens zu einem referenzierenden Knoten

Der zuvor beschriebene Regelsatz enthält die Vorbedingung, dass keine referenzierenden Knoten mit den JPL-Schnittstellenknoten verbunden sein dürfen. Dies ist notwendig, da ansonsten inkonsistente Suchmuster entstehen würden, die zwei referenzierende Knoten enthalten würden (zum einen den vom Nutzer spezifizierten und zum anderen den durch den Regelsatz hinzugefügten), obwohl im Suchmuster nur eine Referenz abgebildet ist. Um dies zu verhindern, muss ein bestehender Referenzknoten in die linke Seite des Regelsatzes mit aufgenommen werden. Dies erfolgt durch den unten dargestellten alternativen Regelsatz. Weiterhin wird die Verbindung zum *Bereichsmarkierungsknoten*, der das importierende Modul umfasst, gelöscht, da der Referenzknoten bereits mit diesem Markierungsknoten verbunden ist.

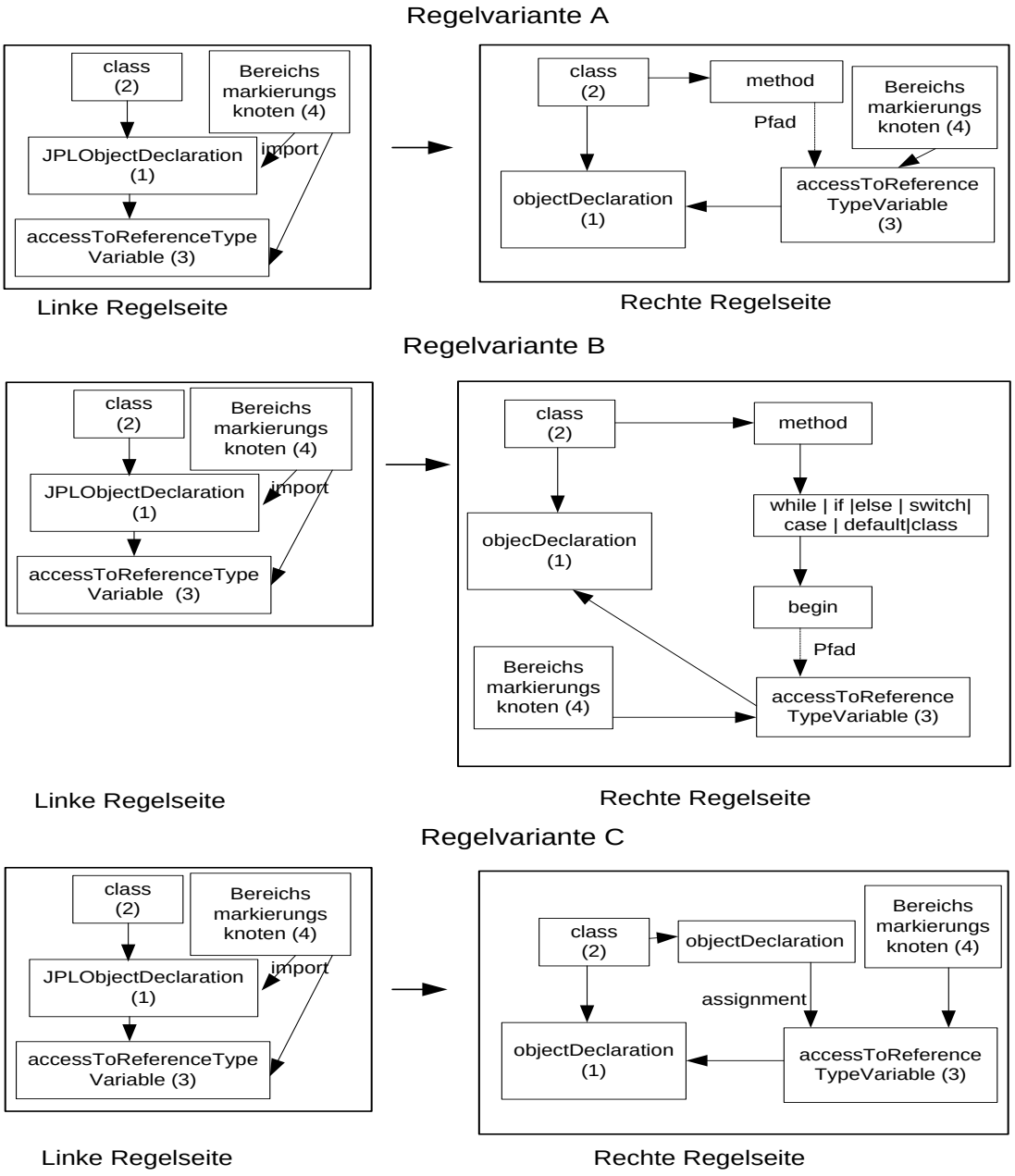


Abbildung 36: Regelvarianten zur Ermittlung des Gültigkeitsbereichs eines Variablenaufrufs bei verbundenem Referenzknoten

Einschränkung: Durch die Ausführung der oben dargestellten Regeln werden Implementierungsvarianten für einen Referenznoten erstellt, dürfen also jeweils auch nur einmal ausgeführt werden. Falls mehrere Referenzknoten mit einem *JPLObjectDeclaration*-Knoten verbunden sind, so wird für die weiteren Referenzen der Gültigkeitsbereich des Aufrufs nicht genauer festgestellt, da die Regeln bei einem Mehrfachaufruf doppelte Strukturen erzeugen würden, die im Suchmuster nicht vorgesehen sind, wie z.B. zwei Knoten vom Typ *method*, obwohl beide Zugriffe nur aus einer Methode heraus erfolgen.

Hinweis: Die oben beschrieben Notwendigkeit, abhängig von den verbunden Referenzknoten verschiedene Regelvarianten einzusetzen, ergibt sich ebenfalls bei der im folgenden Kapitel beschriebenen indirekten Variablenübergabe.

Direkte Übergabe in externe Gültigkeitsbereiche:

Im ersten Schritt wird getestet, ob die Membervariable extern referenziert wird. Hierzu wird die Graphstruktur in NAC 1 als vorgelagertes Suchmuster eingesetzt. Wird für dieses Muster ein Match gefunden, so können alle oben beschriebenen Varianten wieder verwendet werden, wobei in den Varianten A und B die Beziehung zwischen *class*-Knoten und *method*-Knoten auf der rechten Regelseite gelöscht werden müssen und der *method* Knoten mit einem zweiten *class*-Knoten verbunden werden muss. In Variante C wird die Beziehung zwischen *class*- und *objectDeclaration*-Knoten gelöscht und der *objectDeclaration*-Knoten mit einem weiteren Knoten des Typs *class* verbunden. Bei der nachfolgenden Umwandlung des *JPLObjectDeclaration*-Knotens in einen *objectDeclaration*-Knoten muss dieser auf *public* gesetzt werden, da nur öffentliche Variablen direkt exportiert werden können. Durch die oben beschriebenen Regeln kann nun der externe Gültigkeitsbereich, aus dem auf die Membervariable zugegriffen wird, identifiziert werden. Somit ergeben sich weitere 9 strukturell unterschiedliche Muster zu Implementierungsvarianten, die aus dem JPL-Graphen pro Übergabestruktur erstellt werden:

3 mögliche Membervariabletypen * 3 Varianten für Übergaben in externe Bereiche = 9 Implementierungsvarianten.

Übergabetyp: Direkter Aufruf einer lokalen Variablen

Lokale Variablen können nicht direkt extern referenziert werden. Daraus folgt, dass nur interne Referenzen betrachtet werden müssen. Im ersten Schritt muss analysiert werden, in welchem Gültigkeitsbereich die Deklaration erfolgt. Hierzu ist die unten dargestellte Transformation auf dem JPL-Graphen vorzunehmen. Hierbei muss durch eine nachfolgende Abfrage sichergestellt werden, dass auf dem Pfad kein *begin*-Knoten, d.h. kein untergeordneter Gültigkeitsbereich liegt (Regel nicht dargestellt). Falls dies der Fall ist, so muss mit diesem untergeordneten Element die Abfrage wiederholt werden.

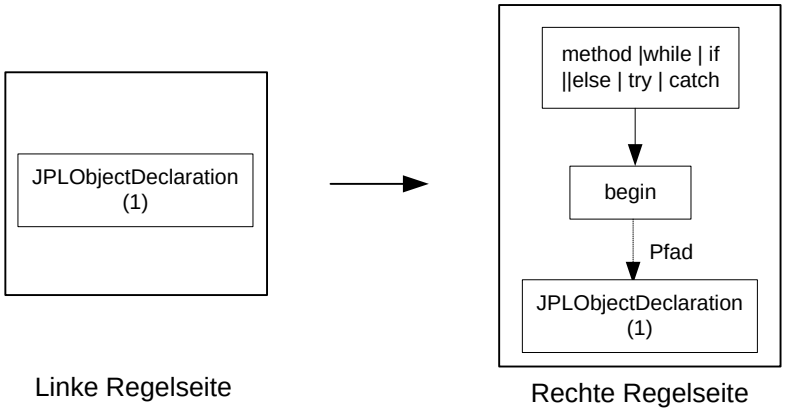


Abbildung 37: Regel zur Suche exportierender lokaler Gültigkeitsbereiche

.Im zweiten Schritt wird nun das Muster um die Referenzierung der Deklaration ergänzt. Folgende Regel wird auf dem JPL Graphen durchgeführt:

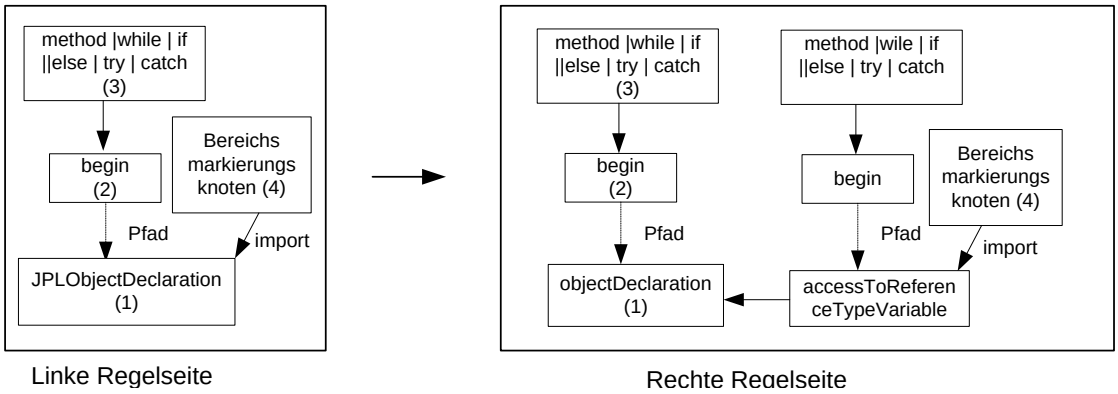


Abbildung 38: Regel zur Suche ex- und importierender lokaler Gültigkeitsbereiche

Die Regel ähnelt der Regel zur Lokalisierung des exportierenden Gültigkeitsbereichs. Hier wird ebenfalls über das sich ergebende Suchmuster die Analyse aller Gültigkeitsbereiche unterhalb der Klassenebene dargestellt, wobei auch hier durch eine nachgelagerte Abfrage geprüft werden muss, ob auf dem Pfad ein *begin*-Knoten liegt und ob der Aufruf sich in einer eingebetteten Gültigkeitsbereichsebene befindet (Regel nicht dargestellt). Über eine Permutation über die Gültigkeitsbereiche (*method*, *while*, *if, else*, *try*, *catch*) und die Erstellung der entsprechenden Muster werden alle möglichen Implementierungsvarianten für die Übergabe lokaler Variablen abgedeckt.

Bedingungen des Regelsatzes:

Vorbedingung: Es existiert ein Knoten vom Typ *JPLObjectDeclaration* im JPL-Graphen, mit dem die Bereichsmarkierung des Moduls, in dem der Schnittstellenknoten im Import definiert wurde, verbunden ist.

Nachbedingungen:

Regel 1: Der Knoten wurde um eine Struktur, die eine Deklaration in einem lokalen Gültigkeitsbereich repräsentiert, ergänzt.

Regel 2: Der Knoten wurde um eine Struktur, die einen Zugriff aus einem lokalen Gültigkeitsbereich heraus repräsentiert, ergänzt. Der Typ *JPLObjectDeclaration* wurde

in *objectDeclaration* geändert. Die Bereichsmarkierung wurde auf die importierende Referenz umgehängt. Die Erstellung des Musters ist damit abgeschlossen.

Der alternative Regelsatz zu Abbildung 38, welcher bei einer bestehenden Verbindung des *JPLObjectDeclaration*-Knotens mit einem referenzierenden Knoten ausgeführt werden muss, ist im Anhang abgebildet. Auch hier gilt, dass bei mehreren bestehenden Verbindungen zu referenzierenden Knoten im JPL-Graphen nur eine Beziehung betrachtet wird.

Durch die oben beschriebenen Regeln wird der JPL-Graph so transformiert, dass bei Anwendung des Musters der jeweiligen Implementierungsvariante die Gültigkeitsbereiche einer direkten Variablenübergabe, d.h. der Bereich der Deklaration und der Bereich, in dem der Zugriff implementiert wurde, identifiziert werden können.

Exkurs:

Direkte Übergabestrukturen, die über JCG-Schnittstellenknoten und AJSDG-Schnittstellenknoten in der JPL-Spezifikation erstellt wurden, werden über eine Verschmelzung der Knoten vom Typ *Declaration* (JCG) bzw. *Anweisungsknoten* (AJSDG) abgeleitet (Regeln s. Anhang 11.2).

Im folgenden Kapitel werden Regeln zur Erstellung von Suchmustervarianten dargestellt, die JPL-Schnittstellenstrukturen in JCG-Strukturen transformieren, über welche verschiedene Typen der indirekten Variablenübergabe erkannt werden können.

5.6. Varianten für die Variablenübergabe durch indirekte Aufrufe

In diesem Kapitel wird die Ableitung der JPL-Schnittstellenstrukturen für die Implementierungsvarianten der indirekten Aufrufe über Graphtransformationsregeln auf dem JPL-Graphen beschrieben. Indirekte Aufrufe sind charaktarisiert durch die Nutzung von Hilfskonstrukten zur Variablenübergabe, wie z.B. die Nutzung von Parametern beim Methodenaufruf.

Zu Beginn der Ableitung des JPL-Musters werden zwei *JPLObject*-Schnittstellenknoten im Import und Export der Modulverbindung durch zwei *JPLObjectDeclaration*-Knoten ersetzt. Dies erfolgt durch eine Änderung der *type* Attribute der entsprechenden JCG-Knoten. Somit ergeben sich zwei Deklarationsknoten, die für die nachfolgend zu erstellenden Suchmuster zu beachten sind, im Gegensatz zu nur einem Knoten bei der zuvor erläuterten direkten Variablenübergabe.

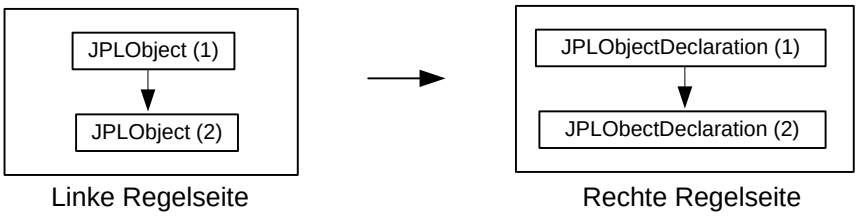


Abbildung 39: Änderung des Typs JPLPrimitive in JPLPrimitiveDeclaration

Die Strukturen der Suchmuster für indirekte Variablenübergaben sind komplexer, als die der direkten Zugriffe, da mehr Elemente betrachtet werden müssen. Im Folgenden wird die Übergabe eines Wertes oder Objekts an eine Methode, die Übergabe an einen Konstruktor, die Rückgabe eines Wertes oder Objekts von einer Methode, die Übergabe von privaten Membervariablen durch Getter/Setter-Methoden und die Übergabe durch eine hierarchisch höher stehende Variable dargestellt.

In den Suchmustern wird der zu übergebende Parameter direkt referenziert. Ausdrücke, wie z.B. Berechnungen, oder die Angabe einer in einer externen Klasse deklarierten Variablen, werden hier nicht betrachtet.

Wie im vorhergehenden Kapitel, so muss auch hier zwischen alleinstehenden *JPLObjectDeclaration*-Knoten und Knoten, die mit Referenzknoten verbunden sind, unterschieden werden. Die folgenden Regeln besitzen die Vorbedingung, dass keine Verbindung zu weiteren Knoten besteht. Die alternativen Regeln für existierende Verbindungen sind im Anhang aufgeführt (S. 216 ff.). Diese Einschränkung gilt nicht für die Übergabevariante: Übergabe durch die *return* Anweisung.

Übergabetyp: Objektvariable wird als Parameter an eine Methode übergeben.

Die Erstellung des Suchmusters für die Übergabe eines Objekts an eine Methode ist in Abbildung 40 beschrieben. Die Verbindung zwischen Export und Import wird über die Kante zwischen Methodenaufruf und Methodenkopf hergestellt. Für eine genauere Analyse des Import- und Export-Gültigkeitsbereichs können die im letzten Abschnitt dargestellten Regeln zur Ermittlung des übergebenden (Export) Gültigkeitsbereichs eingesetzt werden. Der Ziel-Gültigkeitsbereich (Import) ist eine Methode und muss somit nicht ermittelt werden, es sei denn der Methodenname ist für die Analyse erforderlich.

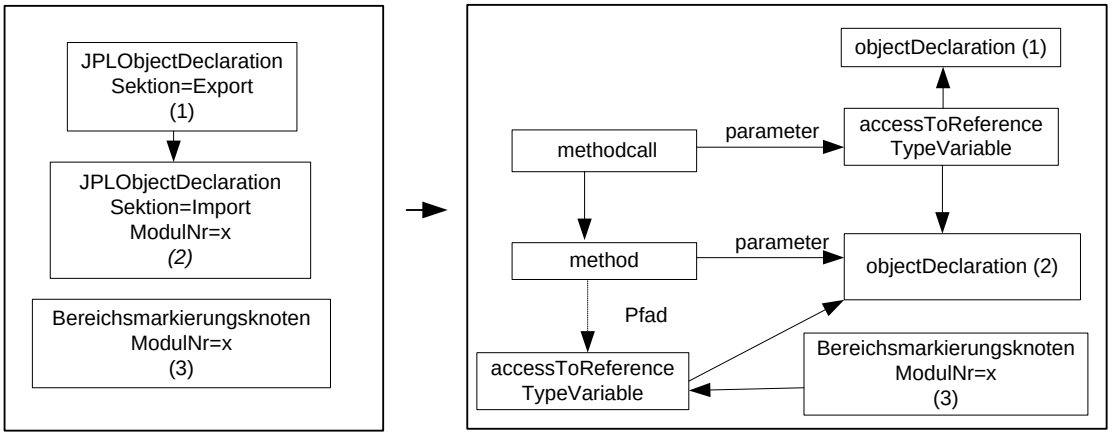


Abbildung 40: Regel zur Erstellung des Suchmusters der Parameterübergabe

Bedingungen des Regelsatzes:

Vorbedingung: Es existieren zwei Knoten vom Typ *JPLObjectDeclaration* im JPL-Graphen. Es existiert ein *Bereichsmarkierungsknoten*, der dem Modul zugeordnet ist, zu dem auch der *JPLObjectDeclaration*-Knoten, dessen Attribut *Sektion* mit *Import* belegt ist, gehört.

Nachbedingung: Es wurde eine Graphstruktur zur Übergabe eines Methodenparameters erstellt. Der Typ *JPLObjectDeclaration* der identifizierten Knoten wurde in *objectDeclaration* geändert. Der Bereichsmarkierungsknoten wurde dem referenzierenden Knoten des Typs *accessToReferenceTypeVariable* zugeordnet, welcher durch einen Pfad mit dem Methodenkopf verbunden ist.

Übergabetyp: Objektvariable wird als Parameter an einen Konstruktor übergeben.

Die Erstellung des Suchmusters für die Übergabe eines Objekts an einen Konstruktor ist in Abbildung 41 dargestellt. Die Verbindung zwischen Export und Import wird über die Kante zwischen ObjektInstanziierung und dem Konstruktorkopf hergestellt. Für eine genauere Analyse des Import- und Export-Gültigkeitsbereichs können die im letzten Abschnitt aufgeführten Regeln zur Ermittlung des übergebenden (Export) Gültigkeitsbereichs eingesetzt werden. Der Ziel-Gültigkeitsbereich (Import) ist ein Konstruktor und muss somit nicht ermittelt werden.

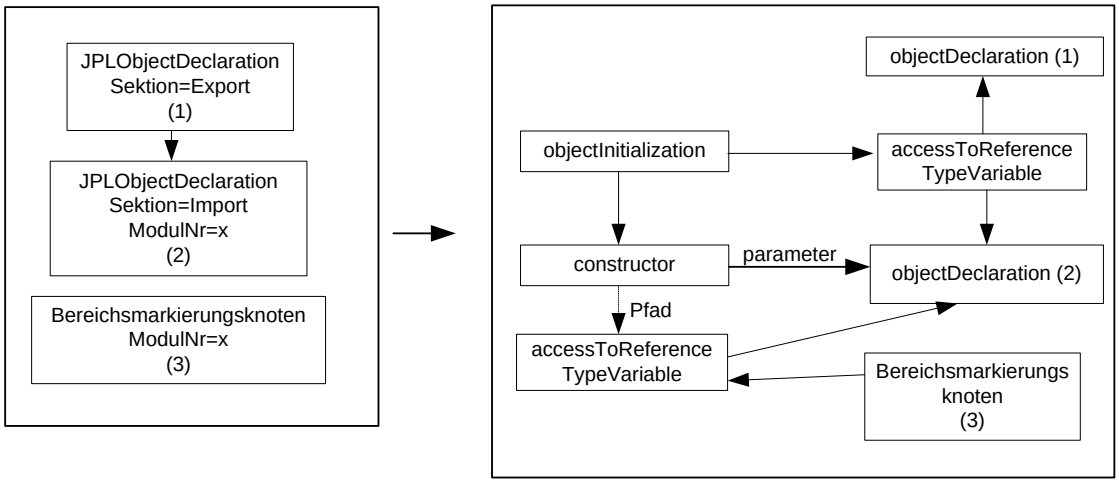


Abbildung 41: Regel zur Erstellung des Suchmusters der Übergabe durch Objektinstanziierung

Bedingungen des Regelsatzes:

Vorbedingung: Es existieren zwei Knoten vom Typ *JPLObjectDeclaration* im JPL-Graphen. Es existiert ein *Bereichsmarkierungsknoten*, der dem Modul zugeordnet ist, zu dem auch der *JPLObjectDeclaration*-Knoten, dessen Attribut *Sektion* mit *Import* belegt ist, gehört.

Nachbedingung: Es wurde eine Graphstruktur zur Übergabe eines Konstruktorenparameters erstellt. Der Typ *JPLObjectDeclaration* der identifizierten Knoten wurde in *objectDeclaration* geändert. Der Bereichsmarkierungsknoten wurde dem referenzierenden Knoten des Typs *accessToReferenceTypeVariable* zugeordnet, welcher durch einen Pfad mit dem Konstruktorkopf verbunden ist.

Übergabetyp: Ein Wert wird als Return-Parameter übergeben.

In Abbildung 42 wird die Werteübergabe einer Methode zu ihren aufrufenden Methoden über die *return* Anweisung dargestellt. Die Verbindung zwischen dem exportierenden Modul und dem importierenden Modul wird über die Kante zwischen dem *return*-Knoten und dem *methodcall*-Knoten realisiert. Während im Export die Objektdeklaration erhalten bleibt, wird der *JPLObjectDeclaration*-Knoten des Imports in einen Knoten vom Typ *methodcall* umgewandelt. Bestehen Verbindungen eines Referenzknotens vom Typ *accessToReferenceTypeVariable* zu dem Knoten vom Typ *JPLObjectDeklaration*, der umgewandelt werden muss, so müssen diese miteinander verschmolzen werden, d.h. die eingehenden Kanten dieses Referenzknotens müssen auf den *methodCall*-Knoten umgeleitet werden. Die Graphregeln hierzu sind im Anhang dargestellt. Hierdurch werden auch AJSDG Knoten, deren ID identisch ist mit dem *accessToReferenceTypeVariable*-Knoten, sowie der *Bereichsmarkierungsknoten*, mit dem *methodCall*-Knoten, assoziiert (Aus diesem Grund muss der importierende *Bereichsmarkierungsknoten* nicht, wie in den Übergabevarianten oben, mit einem Knoten der rechten Regelseite verbunden werden. Falls keine Verbindung eines Referenzknotens besteht ist dies notwendig.). Die Typtransformationen des importierenden *JPLObjectDeclaration-Knoten* sind notwendig, da in dieser Implementierungsvariante der übergebene Wert nicht einer eigenständigen Variablen

wie einer globalen Variablen oder einem Methodenparameter zugewiesen wird, sondern strukturell an den Methodenaufruf übergeht. Um das Muster korrekt erkennen zu können, ist es notwendig, dass über die *return*-Anweisung eine Variable direkt übergeben wird und kein Ausdruck.

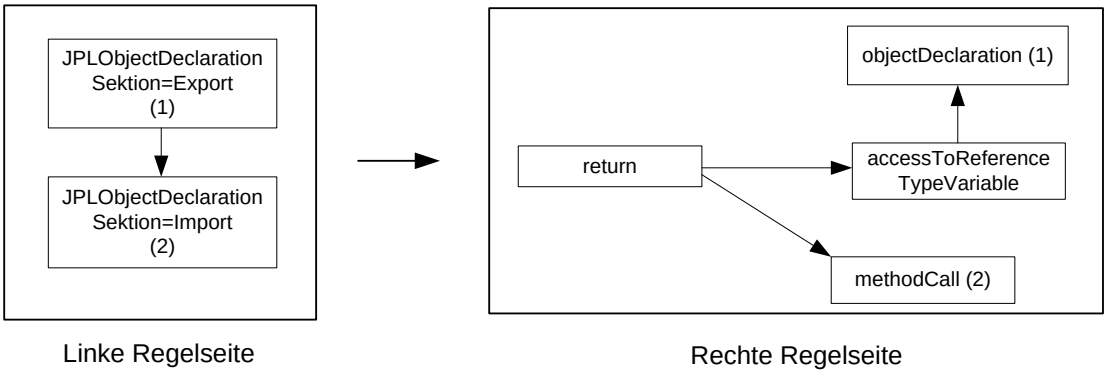


Abbildung 42: Regel zur Erstellung des Suchmusters der Übergabe durch *return*

Bedingungen des Regelsatzes:

Vorbedingung: Es existieren zwei Knoten vom Typ *JPLObjectDeclaration* im JPL-Graphen, die miteinander verbunden sind.

Nachbedingung: Es wurde eine Graphstruktur zur Übergabe im Rahmen eines *return*-Parameters erstellt. Der Typ des *JPLObjectDeclaration*-Knotens, der im Import spezifiziert ist, wurde in *methodcall* geändert. Der Typ des *JPLObjectDeclaration*-Knotens im Export wurde in *objectDeclaration* geändert.

Zu beachtender Sonderfall, falls im Suchmuster eine Beziehung zwischen JCG und AJSDG spezifiziert wurde:

Falls ein Knoten vom Typ *Hilfsknoten-ID* mit dem *JPLPrimitiveDeclaration* Knoten verbunden wurde, müssen die beiden Regeln auf Seite 214, Abbildung 108 und Abbildung 109, angewendet werden, um auch die Beziehung zwischen JCG- und AJSDG-Knoten in das Suchmuster zu übertragen.

Übergabetyp: Aufruf einer Membervariablen über Getter/Setter Methoden.

Nachdem der direkte externe Zugriff nur für öffentliche Membervariablen möglich ist, kann über Getter-/Setter-Methoden auch auf private Membervariablen zugegriffen werden. Die im Folgenden gezeigten Strukturen können aus den Regeln in Abbildung 40 und Abbildung 42 abgeleitet werden.

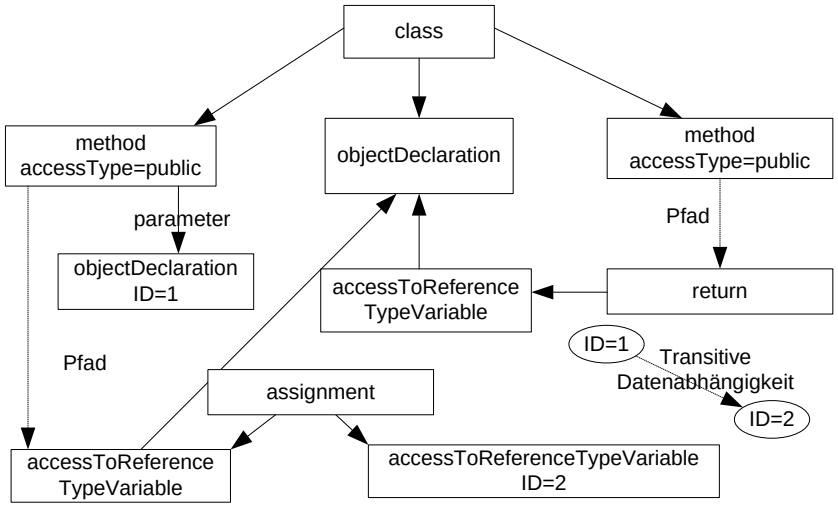


Abbildung 43: Suchmuster der Übergabe durch *Getter/Setter-Methoden*

Die Grundstruktur eines Getter/Setter-Konstrukts wird in Abbildung 43 dargestellt. Links wird die Setter-Methode beschrieben. Diese enthält einen Parameter, dessen Wert an die betrachtete Membervariable übergeben wird. Der Datenfluss über die Datenabhängigkeitsknoten ist notwendig, um die Nutzung von Hilfsvariablen in der Setter-Methode abzudecken. Auf der rechten Seite des Musters wird die Getter-Methode spezifiziert. Aus dieser Grundstruktur resultieren zwei Suchmuster:

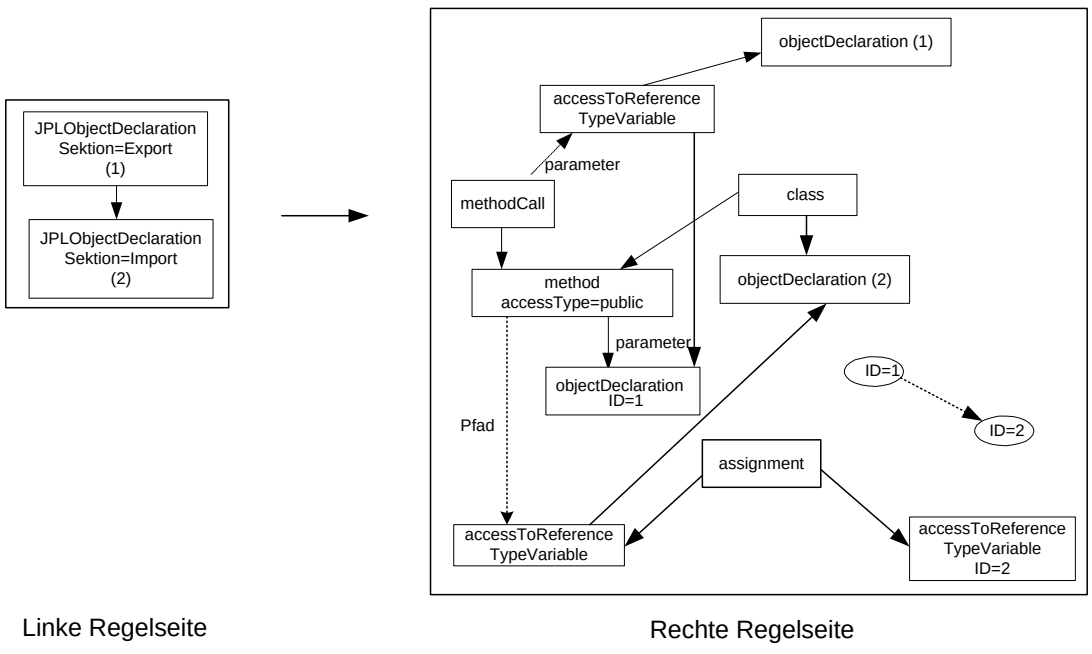


Abbildung 44: Regel zur Erstellung des Suchmusters für eine Setter-Methode

Der zuvor beschriebene Basisgraph für die Setter-Struktur bildet den Ausgangspunkt für die Regel in

Abbildung 44. Hinzu kommt die Struktur des Methodenaufrufs, über den der Wert als Methodenparameter übergeben wird. Die JPL-Schnittstellenknoten werden auf die Membervariable der Klasse, welche die Setter-Methode enthält, und die Deklaration des Parameters, welcher über den Aufruf übergeben wird, gemapped.

Bedingungen des Regelsatzes:

Vorbedingung: Es existieren zwei miteinander verbundene Knoten vom Typ *JPLObjectDeclaration* im JPL-Graphen.

Nachbedingung: Es wurde eine Graphstruktur zur Übergabe im Rahmen einer *Setter*-Struktur erstellt. Der Typ der *JPLObjectDeclaration*-Knoten wurde in *objectDeclaration* geändert.

Abbildung 45 zeigt die Regel zur Ableitung des JPL-Suchmusters für die Implementierungsvariante der Getter-Methode. Hier wird der Knoten vom Typ *JPLObjectDeclaration* im Import zu einem Knoten vom Typ *methodcall* transformiert (zu Details s. Übergabevariante: Wert wird als *return*-Parameter übergeben).

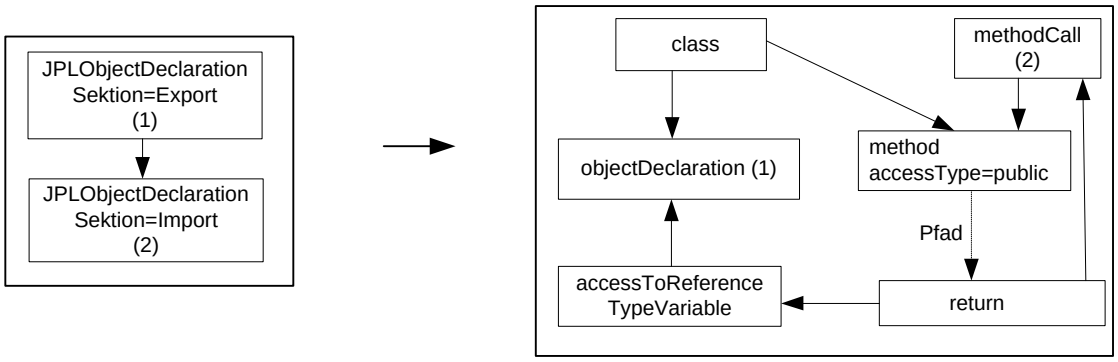


Abbildung 45: Regel zur Erstellung des Suchmusters für eine Getter-Methode

Bedingungen des Regelsatzes:

Vorbedingung: Es existieren zwei miteinander verbundene Knoten vom Typ *JPLObjectDeclaration* im JPL-Graphen.

Nachbedingung: Es wurde eine Graphstruktur zur Übergabe im Rahmen einer *Getter*-Struktur erstellt. Der Typ des *JPLObjectDeclaration*-Knotens, der im Import spezifiziert ist, wurde in *methodcall* geändert. Der Typ des *JPLObjectDeclaration*-Knotens im Export wurde in *objectDeclaration* geändert.

Übergabetyp: Nutzung einer Variablen in einem hierarchisch übergeordneten Gültigkeitsbereich

Eine Übergabevariante, die in der JPL-Schnittstellenstruktur gekapselt ist, umfasst die Übergabe durch eine Variable, die in einem hierarchisch übergeordneten, oder in einem von den Modulen unabhängigen Gültigkeitsbereich deklariert wurde. So kann z.B. eine Membervariable in Methode A belegt und nachfolgend in Methode B ausgelesen werden. Da die Analyse im Rahmen der JPL statisch erfolgt, wird nicht die Reihenfolge der Variablenaufrufe betrachtet, sondern nur die Variablenzugriffe in den Gültigkeitsbereichen.

Vor Ausführung dieser Regel muss der Regelsatz aus Abbildung 32 ausgeführt werden und nicht der Regelsatz aus Abbildung 39, da für diese indirekte Variablenübergabe nur eine Variablendeklaration referenziert wird.

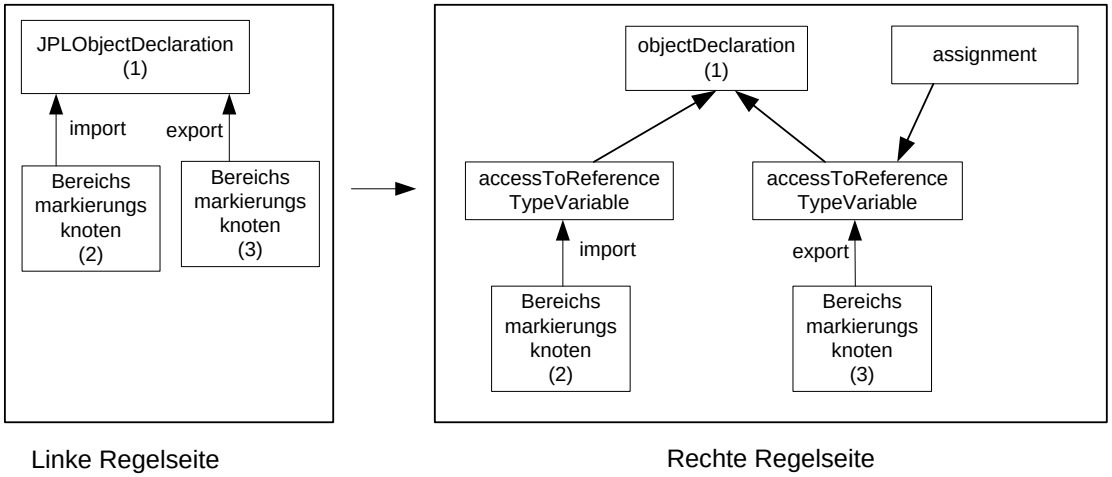


Abbildung 46: Regel zur Erstellung des Suchmusters der Übergabe mittels einer hierarchisch übergeordneten Variablen

Der Gültigkeitsbereich, in dem die Objektdeklaration erfolgt (Membervariable, lokale Variable, etc.) kann ermittelt werden, indem die Mustervariante um die Varianten ergänzt wird, die in Kapitel 5.5 erläutert wurden. Somit stellt diese Übergabevariante eine Erweiterung der in Kapitel 5.5 beschriebenen Muster dar, bzw. kann in diese integriert werden.

Bedingungen des Regelsatzes:

Vorbedingung: Es existiert ein Knoten vom Typ *JPLObjectDeclaration* im JPL-Graphen, der mit zwei Bereichsmarkierungsknoten verbunden ist.

Nachbedingung: Die Verbindungen zwischen dem *JPLObjectDeclaration*-Knoten und den *Bereichsmarkierungs*-Knoten wurden gelöscht und diese neu erstellten *accessToReferenceType*-Knoten zugeordnet. Weiterhin wurde ein *assignment*-Knoten mit der Referenzierung verbunden, die im Export spezifiziert wurde.

Variablenübergaben über *JPLPrimitive, JPLSpecific* und *JPLVariable*

Die zuvor gezeigten Regeln für den Knotentyp *JPLObject* können für die Knotentypen *JPLPrimitive* und *JPLVariable* adaptiert werden, indem der Knotentyp des Deklarationsknotens mit dem geforderten Übergabetyp parametrisiert wird. Um die Regeln für den Typ *JPLSpecific* übernehmen zu können, muss zusätzlich das Attribut DataType mit dem geforderten Typ belegt werden (*long, int, double*). Deklarationsknoten für primitive Datentypen werden über den Knotentyp *accessToPrimitiveDeclaration* referenziert.

In diesem Kapitel wurden die Regelsätze zur indirekten Variablenübergabe erläutert, die durch die JPL-Schnittstellenstrukturen gekapselt werden. Durch Anwendung dieser Regelsätze auf einen JPL-Graphen werden die JCG-basierten Suchmuster zu den Implementierungsvarianten für indirekte Variablenübergaben erstellt.

Während sich die vorhergehenden Kapitel mit der Erstellung von Suchmustern zu den verschiedenen Implementierungsvarianten für den vollständigen JPL-Graphen beschäftigt haben, wird im Folgenden der Prozess zur Ableitung der JPL-Schnittstellenstrukturen für die Variante der Mustersuche, in der die Modulsektionen separat betrachtet werden, beschrieben.

5.7. Erstellung der Datei zur Mustersuche bei separater Sektionsbetrachtung

In diesem Kapitel wird der Prozess der Ableitung von JPL-Suchmustern für die Variante der Mustersuche dargestellt, welche die Graphen der Sektionen separat voneinander betrachtet (die in Kapitel 5.3. beschriebene Variante zwei). Dies bedeutet, dass der JPL-Graph sektionsbasiert in mehrere Teilgraphen zerlegt wird (Schritte 1 – 6), die während des Ablaufs der Mustersuche einzeln identifiziert werden müssen. Weiterhin wird ein Suchmuster erstellt, welches die Matches der Teilmuster zusammen erkennt, um somit das Gesamtmuster zu identifizieren (Schritt 7). Die Einbettung der im Folgenden betrachteten Regeln zur Musteridentifikation in den Gesamtablauf der Mustersuche ist in den Kapiteln 5.9. und 5.10 beschrieben.

Prozess der Musterzerlegung auf dem abgeleiteten JPL-Graphen (die JPL-Schnittstellenstrukturen wurden zuvor in JCG-Elementstrukturen abgeleitet):

1. Verbinde die Bereichsmarkierungsknoten mit den zugehörigen JCG- und AJSDG- Knoten.
2. Markiere pro Modul die Knoten der Körpersektion mit einem Markierungsknoten.

   Im Folgenden wird die initiale Regel zur Markierung der Körpersektion dargestellt (Erstellung des Markierungsknotens). Falls bereits ein Markierungsknoten für diesen Bereich existiert, werden die weiteren JCG-Knoten durch eine Kante mit diesem verbunden.

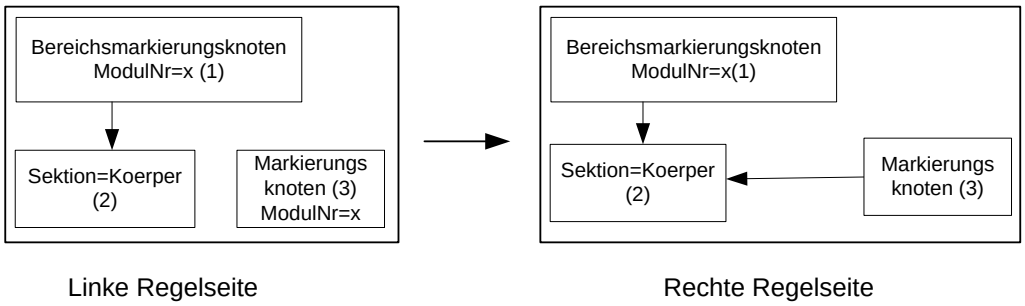


Abbildung 47: Regel zur Erstellung des Teilmusters, welches den Graphen der Körpersektion umfasst

3. Verbinde die Übergabestrukturen mit einem Schnittstellenmarkierungsknoten. Die verschiedenen Übergabevarianten auf dem JPL-Graphen wurden in den vorhergehenden Kapiteln beschrieben (bsp. Kapitel 5.5 Abbildung 35, Kapitel 5.6. Abbildung 40 bis Abbildung 42 und Abbildung 44 bis Abbildung 46, die rechte Seite der dargestellten Regen) und müssen für diesen Schritt um Bereichsmarkierungsknoten ergänzt werden, die jeweils mit einem Knoten des exportierenden und des importierenden Teils der Übergabevariante verbunden werden. Falls Knoten der Übergabestruktur mit Knoten aus der Körpersektion verbunden sind, so werden die direkt verbundenen Knoten aus der Körpersektion in die Übergabestruktur mit aufgenommen und mit dem Schnittstellenknoten markiert. Falls ein oder mehrere Knoten aus der Identifikator-Sektion mit der Übergabestruktur verbunden sind, so sind auch diese dem Muster hinzuzufügen.

   Beispiel zur Markierung einer Übergabevariante (eine Membervariable wird direkt referenziert):

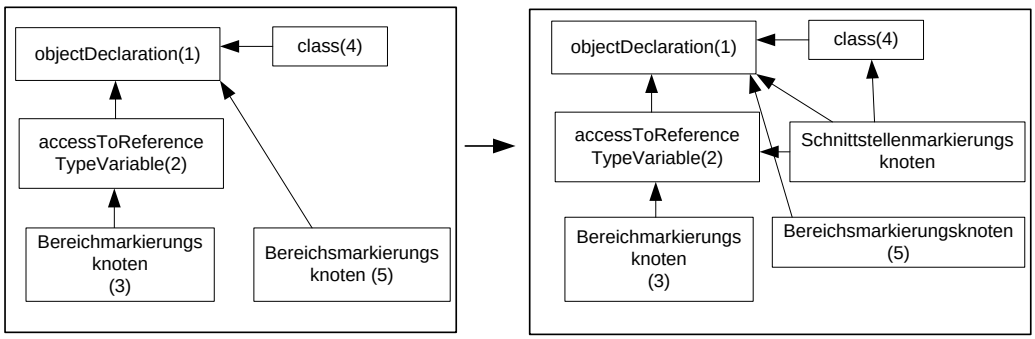


Abbildung 48: Regel zur Erstellung des Teilmusters, welches den Graphen zur Übergabestruktur umfasst

4. Nachdem alle Knoten markiert wurden, wird der Graph in einer XML-Datei gespeichert.

Einfügen der Regeln zur Identifikation der Graphstrukturen in die Datei zum Ablauf der Mustersuche:

5. Erstelle für die Graphen in der Körpersektion pro Modul eine Regel, deren linke Regelseite den Graph des Modulkörpers enthält und deren rechte Regelseite bei einem Match die Knoten mit einem Markierungsknoten verbindet. Es werden alle Körpersektionen des abgeleiteten JPL-Graphen sequentiell betrachtet. Die entsprechenden Graphen können ermittelt werden, indem über Graphregeln alle Knoten und die mit diesen Knoten verbundenen Kanten gelöscht werden, welche nicht mit dem *Bereichsmarkierungsknoten* welcher die aktuell betrachtete Körpersektion markiert, verbunden sind. Der verbleibe Graph wird auf die rechte Reglseite der neuen Regel kopiert kopiert. Danach wird per Graphregel der Markierungsknoten und die zugehörigen Kanten gelöscht und der nun verbleibende Graph auf die linke Reglseite kopiert und ein vollständiger Match zum Graphen auf der rechten Regelseite erzeugt. Falls sich aufeinander beziehende JCG-Hilfsknoten und AJSDG Knoten im Graphen enthalten sind, so müssen die identischen IDs durch identische Variablen ersetzt werden.

6. Erstelle für jede Übergabestruktur eine Regel. Auf der linken Seite wird die Übergabestruktur ohne Schnittstellenmarkierungsknoten abgebildet und auf der rechten Regelseite wird die Struktur (vollständig gemapped) zusammen mit dem Schnittstellenmarkierungsknoten erzeugt. Dies bedeutet, es wird der Teilgraph eingefügt, welcher in Schritt 3 erzeugt wurde. Es werden alle Übergabestrukturen des abgeleiteten JPL-Graphen sequentiell betrachtet. Die entsprechenden Graphen können ermittelt werden, indem über Graphregeln alle Knoten und die mit diesen Knoten verbundenen Kanten gelöscht werden, welche nicht mit dem Schnittstellenmarkierungsknoten, welcher die aktuell betrachtete Körpersektion markiert, verbunden sind. Der verbleibe Graph wird auf die rechte Reglseite der neuen Regel kopiert kopiert. Danach wird per Graphregel der Schnittstellenmarkierungsknoten und die zugehörigen Kanten gelöscht und der nun verbleibende Graph auf die linke Reglseite kopiert und ein vollständiger Match zum Graphen auf der rechten Regelseite erzeugt.

   Während zuvor die Schnittstellenstrukturen ausgehend vom abgeleiteten JPL-Graphen ermittelt wurden, ist es auch möglich, diese unabhängig hiervon aus einem Katalog von Übergabestrukturen zu entnehmen und in den Regelsatz zur Mustersuche einzufügen. Das Attribut *ModulNr* der Bereichsmarkierungsknoten, die jeweils mit dem importierenden oder exportierenden Teil der Übergabestruktur verbunden sind, muss hierzu anhand der JPL-Schnittstellenknoten parametrisiert und die JPL-Knoten nachfolgend gelöscht werden.

   Einschränkung: Falls Übergabestrukturen mit Strukturen in der Körpersektion oder der Identifikatorsektion verbunden sind, so werden diese Beziehungen bei diesem Vorgehen nicht abgebildet.

Erstellung des Gesamtmusters, welches die verschiedenen Matches der Teilgraphen miteinander kombiniert:

7. Folgende Regeln setzen auf dem JPL-Graphen auf:

    1. Initial werden Bereichsmarkierungsknoten für die zu betrachtenden Module erstellt.
    2. Für Schnittstellenstrukturen werden zwei JCG- oder AJSDG-Knoten erstellt, die jeweils mit einem *Bereichsmarkierungsknoten* der beiden verbundenen Module und einem Schnittstellenmarkierungsknoten verbunden werden (die Regel ist im Anhang auf Seite 218, Abbildung 119 dargestellt).
    3. Pro Körpersektion ist ein JCG-Knoten, oder, bei ausschließlicher Verwendung von AJSDG-Syntaxelementen im Suchmuster, ein AJSDG-Knoten zu erstellen, die mit *Bereichsmarkierungsknoten* und Markierungsknoten der jeweiligen Module (s. Punkt 1) verbunden werden. Falls in einem Modul eine Verbindung eines JPL-Schnittstellenknotens mit einem Knoten der Körpersektion spezifiziert wurde, so wird kein separater JCG-Knoten für dieses Modul erstellt, sondern der Markierungsknoten an den bereits existierenden und mit einem Schnittstellenknoten verbundenen JCG-Knoten des jeweiligen Bereichs gehängt (Regeln für die Markierung des Teilgraphen sind im Anhang auf Seite 219, Abbildung 120 (keine Schnittstellenverbindung) und Abbildung 121 (Schnittstellenverbindung zum Export) dargestellt).
    4. Im nächsten Schritt wird das Gesamtmuster konsolidiert, indem Mehrfachverbindungen zu Markierungsknoten, bzw. zu Markierungsknoten und Schnittstellenknoten, auf nur einen JCG-Knoten gerichtet werden.
    5. Abschließend werden alle Knoten, die nicht zum Gesamtmuster gehören gelöscht, und das Muster in einer Datei gespeichert.

Einfügen der Regel zur Identifikation des Gesamtmusters in die Datei zum Ablauf der Mustersuche:

    6. Die Regel zur Suche nach dem Gesamtmuster wird als zuletzt auszuführende Regel eingefügt. Die linke Regelseite enthält das Gesamtmuster und die rechte Regelseite einen vollständigen Match dieses Musters, ergänzt um einen Ergebnisknoten. Falls ausgeschlossen werden soll das Teilmuster der Körpersektionen in überlappenden Bereichen identifiziert werden, muss die Regel um NACs für alle Bereichsmarkierungsknotenkombinationen ergänzt werden, die verhindern, dass zwei Bereichsmarkierungsknoten auf identische Knoten referenzieren.

Über einen Match des oben erläuterten Musters wird das Gesamtmuster im Rahmen des Regelablaufs zur Mustersuche identifiziert. Während die Markierungsknoten und die Schnittstellenmarkierungsknoten durch die Ausführung der zuvor dargestellten Regeln in den Quellcodegraph eingefügt werden, wird der Bereichsmarkierungsknoten durch

Regeln eingefügt, über die der zu betrachtende Gültigkeitsbereich identifiziert wird (s. Kapitel 5.9. und 5.10).

In diesem Kapitel wurde der Prozess der Zerlegung eines JPL-Graphen (Suchmusters) in sektionsbasierte (Körpersektion und Schnittstellensektionen) Teilgraphen und deren Einbettung in den Regelsatz zur Mustersuche betrachtet. Weiterhin wurde gezeigt, wie der Graph zur Identifikation des Gesamtmusters erzeugt wird. Hiermit ist die Erstellung der Regelmenge für die Identifikation der Teilmuster bei separater Sektionsbetrachtung abgeschlossen.

Eine umfangreiche Darstellung des vollständigen Ablaufs der Mustersuche wird in den Kapiteln 5.9. und 5.10 gegeben. Zuvor wird die Erstellung der Transitiven Hülle durch Graphregeln erläutert, die auf dem AJSDG aufsetzen. Damit ist die Graphrepräsentation des Quellcodes, auf dem die Suchmuster aufsetzen, vollständig beschrieben.

5.8. Transitive Hülle

In diesem Kapitel wird die Erstellung der transitiven Hülle auf einer AJSDG-Struktur dargestellt. Werden in einem JPL-Muster eine oder mehrere transitive Kanten spezifiziert, so ist es notwendig, die Graphrepräsentation des Quelltextes, auf dem die Suche durchgeführt wird, um die transitive Hülle zu ergänzen. Weiterhin ist die transitive Hülle notwendig, um die Suche nach dem Element *Pfad* zu realisieren. Die Spezifikation des AJSDG, auf dem die transitive Hülle aufsetzt, wurde in Kapitel 3.3. beschrieben, bzw. es wurde auf die entsprechenden Arbeiten verwiesen. Die Eigenschaften der transitiven Hülle werden in Kapitel 3.4. dargestellt.

Die Grundlage des im Folgenden vorgestellten Verfahrens bildet die im Rahmen des Warshall-Algorithmus [War62] formulierte Eigenschaft, welche besagt, dass für 3 Knoten *a,b,c* in einem Graphen gilt:

> *Wenn Knoten b von Knoten a aus erreichbar ist und Knoten c von Knoten b aus erreichbar ist, dann ist ebenfalls Knoten c von Knoten a aus erreichbar.*

Während der Algorithmus von Warshall im Weiteren auf einer Inszidenzmatrixdarstellung des Graphen realisiert wird, setzt der Algorithmus in diesem Abschnitt auf der Graphdarstellung des AJSDG auf. Die Einführung einer Matrix würde einen Modellbruch bedeuten, welcher somit vermieden wird.

Im ersten Schritt werden alle direkten Nachbarknoten aller AJSDG-Knoten, die bereits über eine Datenabhängigkeitskante verbunden sind, zusätzlich mit einer transitiven Datenabhängigkeitskante verbunden. Um die transitive Hülle methoden- und klassenübergreifend aufzuspannen, muss in der Regel auch die Parameterkante berücksichtigt werden.

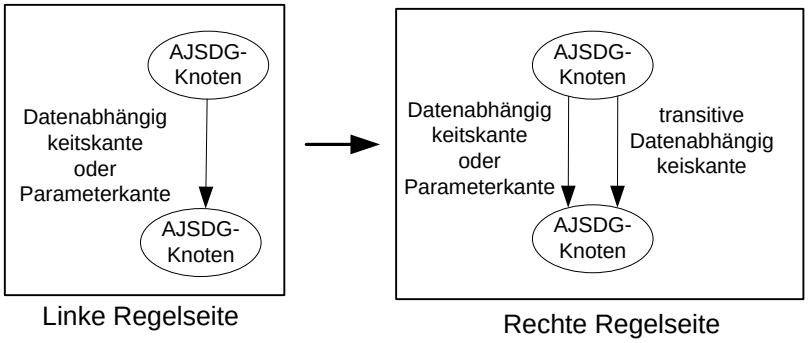


Abbildung 49: Regel 1 zur Erstellung der transitiven Hülle für Datenabhängigkeiten

Im zweiten Schritt werden alle weiteren erreichbaren AJSDG-Knoten durch transitive Kanten verbunden. Hierbei werden, sobald eine Abhängigkeit zwischen zwei Knoten über eine transitive Kante identifiziert wird, auch alle vom Zielknoten der Kante aus erreichbaren Knoten mit dem Quellknoten verbunden.

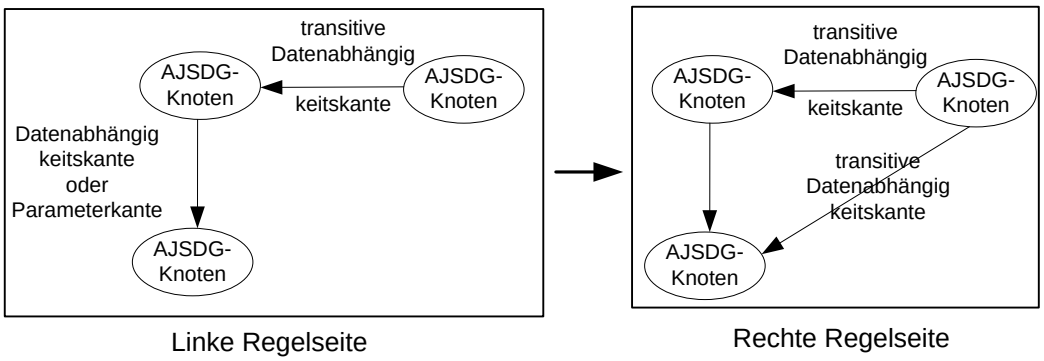


Abbildung 50: Regel 2 zur Erstellung der transitiven Hülle für Datenabhängigkeiten

Bedingungen des Regelsatzes:

Vorbedingung: Die AJSDG Repräsentation des Quellcodes enthält Datenabhängigkeiten.

Nachbedingung: Über alle Datenabhägigkeitsbeziehungen wurde die transitive Hülle erstellt.

Durch die oben abgebildeten zwei Graphregeln wird die vollständige transitive Hülle über die Datenabhängigkeiten der Quellcodedarstellung generiert. Die Erstellung der transitiven Hülle für Kontrollabhängigkeiten erfolgt äquivalent. In den Regeln muss nur die Typisierung der *Datenabhängigkeitskante* in *Kontrollabhängigkeitskante* und die *transitive Datenabhängigkeitskante* in *transitive Kontrollabhängigkeitskante* geändert werden.

Realisierungsvariante: Zur Optimierung der Regelausführung ist es möglich, die transitive Hülle nicht über dem gesamten Quelltext, sondern nur über die für die Mustersuche relevanten Teilbereiche aufzuspannen. Hierzu muss evaluiert werden, von welchen Elementen (AJSDG-Knoten und mit diesen verbundene JCG-Knoten) im JPL-Graphen eine transitive Kante ausgeht. Alle Knoten in der Quellcoderepräsentation, auf denen diese Knoten des JPL-Graphen einen Match erzeugen, werden nachfolgend identifiziert und es wird eine transitive Kante zu den mit diesen Knoten nachfolgend

verbundenen AJSDG- Knoten gezogen. Im zweiten Schritt wird die in Abbildung 50 dargestellte Regel ausgeführt.

Bevor diese Ausführungsvariante gewählt wird, gilt es einzuschätzen, ob der zusätzliche oben beschriebene Aufwand zur Suche der initialen AJSDG-Knoten nicht die Aufwandseinsparung, welche dadurch gewonnen wird, dass nur ein Teil des Codes betrachtet wird, wieder aufwiegt.

Seiteneffekt der Variante: Die Ausführung der Suche nach dem JPL-Syntaxelement *Pfad* setzt auf der transitiven Hülle der Kontrollabhängigkeiten auf (s. Abschnitt 5.9), so dass diese Suche evtl. nicht durchgeführt werden kann, falls die transitive Hülle nur für Codefragmente erstellt wird.

Nachdem die Erstellung der transitiven Hüllen beschrieben und somit alle Elemente sowohl der Quellcoderepräsentation als auch der zu suchenden Mustervarianten erläutert wurden, wird im nächsten Abschnitt die Durchführung der Mustersuche dargestellt.

5.9. Ausführung der Mustererkennung für ein einzelnes JPL-Modul

Die folgende Sequenz beschreibt den Prozess der Mustererkennung basierend auf der Graphrepräsentation des Quellcodes und der aus dem JPL-Graphen abgeleiteten Mustervarianten für ein einzelnes Modul. Durch das im Folgenden dargestellte Verfahren werden beide in Kapitel 5.3. beschriebenen Varianten der Musterableitung und –suche erfasst. Für die Durchführung der Mustersuche nach der Variante *Mustersuche bei separater Betrachtung der einzelnen Sektionen* sind alle unten beschriebenen Prozessschritte notwendig. Der Ablauf der Suche für die Variante *Suche nach vollständigen Mustervarianten* fasst die Schritte 2, 3 und 4 zusammen, da hier das vollständige Muster einer Implementierungsvariante durch nur eine Regel gesucht wird.

Definition: Eine Struktur im Quellcode wird als gültiger *Match* zu einem JPL-Muster bewertet, wenn alle folgenden Bedingungen erfüllt sind:

1. Der spezifizierte Gültigkeitsbereich wird gefunden.
2. Die innerhalb der Gültigkeitsbereiche spezifizierten Strukturen werden gefunden.
3. Die spezifizierten Schnittstellenstrukturen werden gefunden.
4. Es wird eine gültige Kombination von 2 und 3 gefunden, d.h. die Schnittstellenstrukturen beziehen sich auf einen Gültigkeitsbereich, in dem die in der Körpersektion spezifizierten Graphen gefunden wurden.

Prozessschritte des Ablaufs der Mustersuche:

1. Im ersten Schritt wird die Identifikator-Sektion des Musters analysiert. Die Elemente in dieser Sektion grenzen den Typ des Gültigkeitsbereichs ein, z.B. ob es sich um eine Klasse, eine Methode oder eine Schleife handelt. Das hier spezifizierte Element zur Bestimmung des Gültigkeitsbereichs (*class, method, while*, etc.) wird im Quellcodegraph gesucht und durch einen Knoten vom Typ

Bereichsmarkierungsknoten markiert. Der gefundene Gültigkeitsbereich im Quellcode darf noch nicht markiert sein. Alle Knoten innerhalb des Gültigkeitsbereichs des Blocks werden mit diesem Markierungsknoten verbunden, so dass in den folgenden Regeln identifiziert werden kann, welche Elemente zu dem entsprechenden Bereich gehören, und nur diese betrachtet werden. Falls kein Match im Quellcodegraphen gefunden wird, so wird ein Fehlerknoten erstellt und der Mustersuchprozess wird abgebrochen. Der Regelsatz für diesen Schritt ist im Anhang aufgeführt (S. 206 ff).

2. Im markierten Bereich wird die in der Körper Sektion spezifizierte Struktur gesucht. Die Erstellung der Regeln, über welche die Suche durchgeführt wird, beschreiben Kapitel 5.7., Punkt 2 und 5. Wird kein gültiges Muster gefunden, erfolgt ein Rücksprung zu Schritt 1. Während die Elemente des JCG, AJSDG und der transitiven Abhängigkeiten direkt auf der Quelltextrepräsentation gesucht werden können, muss für die Identifikation eines *Pfades*, der im JPL-Graph spezifiziert wurde, eine Regelfolge mit diesem Muster verknüpft werden (zur Erläuterung der Graphregeln s. folgenden Abschnitt). Dies bedeutet, dass zuerst untersucht wird, ob das zu suchende Muster einen *Pfad* enthält. Falls dies der Fall ist, wird die transitive Hülle für den Kontrollfluss auf der AJSDG-Graphstruktur des zu analysierenden Quellcodes erstellt. Danach wird nach Elementen gesucht, welche durch einen Pfad verbunden sein müssen. Die möglichen Anfangs- und Endelemente des Pfades im Quellcode werden markiert, die Regeln zur Pfadsuche ausgeführt und die gefundenen Pfade in der Quelltextrepräsentation eingetragen. Danach wird das vollständige Muster der Körpersektion des JPL-Graphs gesucht und die gefundenen Strukturen markiert. Falls keine Struktur gefunden wird, erfolgt ein Rücksprung zu Punkt 1.

3. Nun werden die Übergabestrukturen im Quelltextgraph über die Regeln gesucht, deren Erstellung in Kapitel 5.7. Punkt 3 und 6 beschrieben wurde. Falls in den Im-/Export-Schnittstellen AJSDG- und/oder JCG-Elemente spezifiziert wurden, so werden zusätzlich Regeln ausgeführt, welche prüfen, ob die hier spezifizierten Elemente im markierten Gültigkeitsbereich gefunden werden. Die im Import spezifizierten Elemente dürfen nicht gefunden werden, die im Export spezifizierten Elemente müssen gefunden werden. Die gefundenen Schnittstellenstrukturen werden im Quellcodegraphen markiert.

   Wird im Quellcode keine Import- oder Exportstruktur gefunden, so wird zu Schritt 1 zurückgesprungen.

4. Es werden alle Elemente der JPL-Struktur im Quellcode zusammen identifiziert und ein Ergebnisknoten wird erstellt. Weiterhin wird, falls das Muster JPL-Schnittstellenknoten enthält, angegeben, welcher Übergabetyp gefunden wurde.

Im Folgenden werden die zuvor dargestellten Prozessschritte im Rahmen einer algorithmischen Beschreibung detailliert. Dieser Algorithmus beschreibt den Regelablauf der Mustersuche für die Variante *Mustersuche bei separater Betrachtung der einzelnen Sektionen*. Er nutzt sowohl allgemeine Regeln, die zur Identifizierung von Gültigkeitsbereichen verwendet werden und im Anhang beschrieben sind (Schritte 1 bis 3), als auch suchmusterspezifische Regeln, die Ergebnis der Ableitung des JPL-Graphen sind, die in den Kapiteln 5.3. bis 5.7. erläutert wurden (Schritte 4 bis 11).

Algorithmus im Detail:

1. Finde das Muster aus der Identifikatorsektion und markiere im Codegraph das Element, welches den Anfang des Gültigkeitsbereichs bildet. Dieses Element darf noch keine Markierung besitzen.
2. Falls das Muster nicht gefunden wird, erstelle einen Ergebnisknoten mit der Meldung „Suchmuster nicht gefunden“ und beende den Algorithmus.
3. Verbinde alle Elemente innerhalb des Gültigkeitsbereichs im Codegraph mit einem Bereichsmarkierungsknoten und markiere alle Elemente, die nicht im Bereich gefunden werden dürfen.
4. Enthält das Muster in der Körpersektion eine transitive Hülle oder einen Pfad?
5. Falls ja, erstelle den AJSDG und die entsprechende transitive Hülle auf dem Quelltextgraphen.
6. Falls das Suchmuster einen *Pfad* enthält, suche im Quelltext mögliche Quell- und Zielelemente und erstelle die zugehörigen Pfadkanten.
7. Suche das Muster der Körpersektion. Mindestens ein gefundenes Element darf nicht markiert sein. Falls kein Muster gefunden wird, lösche alle Kanten zum Bereichsmarierungsknoten sowie den Knoten selbst und gehe zu 1.
8. Markiere die Elemente des gefundenen Musters eindeutig mit einem Markierungsknoten. Ziel ist es, dass dieses Muster bei weiteren möglichen Suchvorgängen nicht mehr betrachtet wird.
9. Enthält die Import- und/oder Exportsektion Elemente? Falls nein gehe zu 12.
10. Falls JCG- und/oder AJSDG-Strukturen in den Schnittstellensektionen definiert sind (hierzu zählen auch die Strukturen, welche aus den JPL-Knoten abgeleitet wurden), suche diese zusammen mit den evtl. hiermit verbundenen Knoten aus der Körpersektion im Quellcode. Die Muster hierzu werden unter Anwendung der Regeln in Kapitel 5.7 erstellt. Die Knoten der gefundenen Übergabestruktur werden mit Schnittstellenmarkierungsknoten markiert. Wird kein Muster gefunden, gehe zu 7.
11. Prüfe, ob das Gesamtmuster gefunden wird. Hierzu wird ein Suchmuster erstellt (s. Kapitel 5.7., Punkt 7), das für jede zu suchende JPL-Schnittstellenstruktur einen Schnittstellenmarkierungsknoten enthält, der mit einem JCG-Knoten verbunden ist. Weiterhin müssen die JCG-Knoten mit dem *Bereichsmarkierungsknoten* des Moduls verbunden werden, um sicherzustellen, dass die Schnittstellenstruktur den korrekten, d.h. den in der Identifikatorsektion spezifizierten Bereich, erfasst. Falls eine Verbindung zwischen einer Schnittstellenstruktur und dem Graphen der Körpersektion spezifiziert wurde, so muss der JCG-Knoten zu diesem Schnittstellenknoten auch mit einem Markierungsknoten verbunden sein. Wird kein Muster gefunden, gehe zu 7.

Beispiel: Folgendes Muster wird zur Gesamtprüfung eines Musters, welches zwei JPL-Übergabestrukturen enthält, genutzt. Eine Übergabestruktur ist mit einem Knoten der Körpersektion des Moduls verbunden.

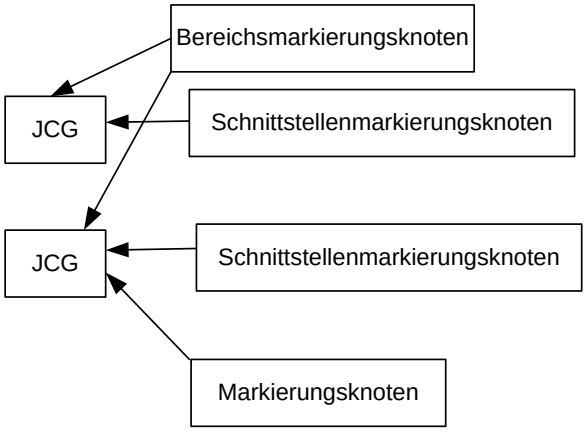


Abbildung 51: Muster zur Identifikation des Gesamtmusters

12. Falls kein Match gefunden wurde lösche alle Schnittstellenmarkierungsknoten und Markierungsknoten und gehe zu 7.
13. Das Muster wurde gefunden. Ein Ergebnisknoten mit dem Namen der gefundenen Implementierungsvariante wird erstellt.

Im Folgenden wird der Algorithmus an einem Beispiel verdeutlicht.

Das Beispiel unten demonstriert ein JPL-Muster, welches folgende Anforderung repräsentiert: „In Methode *VarABelegen* wird die Variable *a* mit dem Wert einer Variablen belegt, welche in die Methode importiert wird (d.h. eine nicht lokale Variable). Die Belegung erfolgt innerhalb einer *if*-Bedingung.“

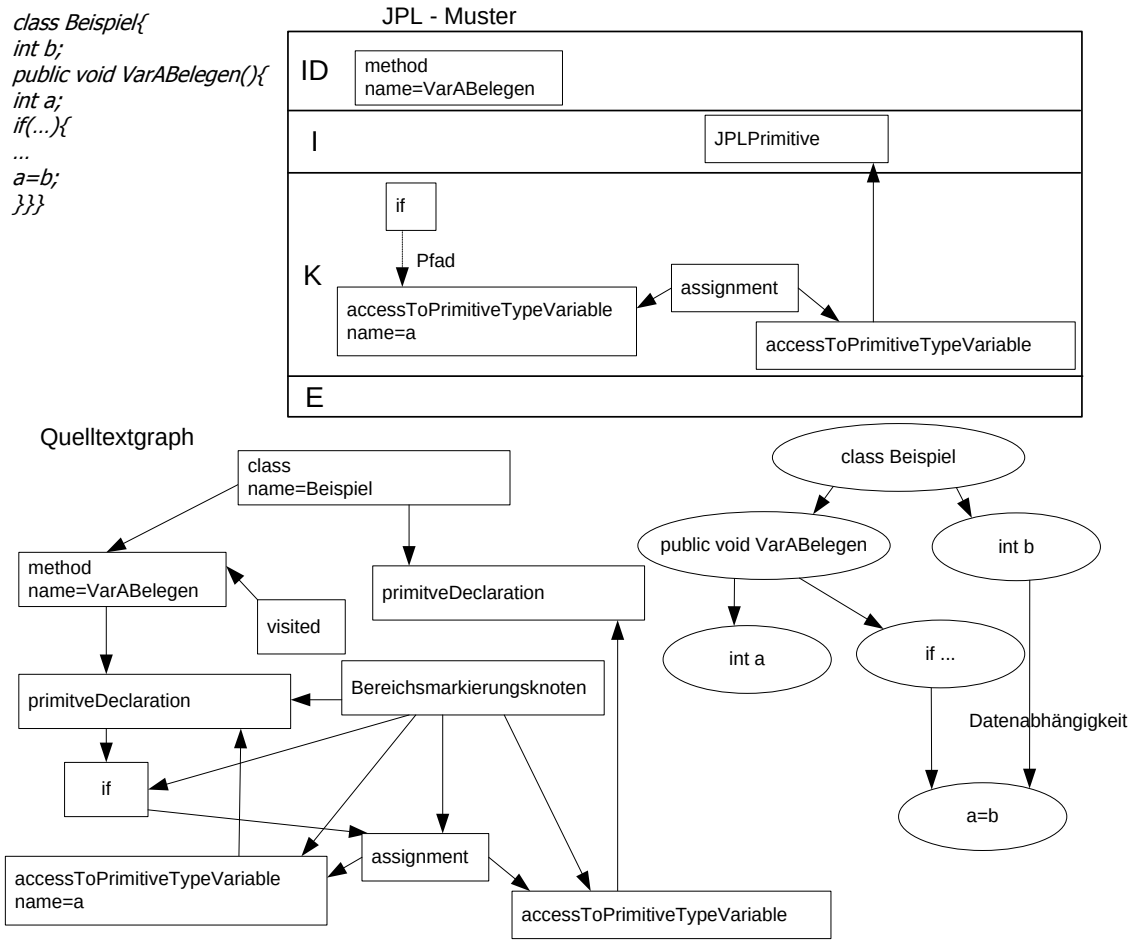


Abbildung 52: Graph eines Einzelmusters nach Schritt 5 des Algorithmus

Der Quelltextgraph enthält den JCG zum links Quellcode. Der auf Grund des Methodenknotens in der Identifikatorsektion gefundene Gültigkeitsbereich wurde markiert (Schritt 3 im Algorithmus). Weiterhin wird der AJSDG mit Kontrollabhängigkeiten und Datenabhängigkeiten abgebildet (Schritt 5 im Algorithmus).

Anmerkung: Die *if*-Bedingung und die hiernach folgenden Ausdrücke wurden aus Gründen der besseren Übersichtlichkeit nicht weiter ausformuliert. Weiterhin wurden die *begin* und *end* Knoten der Gültigkeitsbereiche nicht dargestellt.

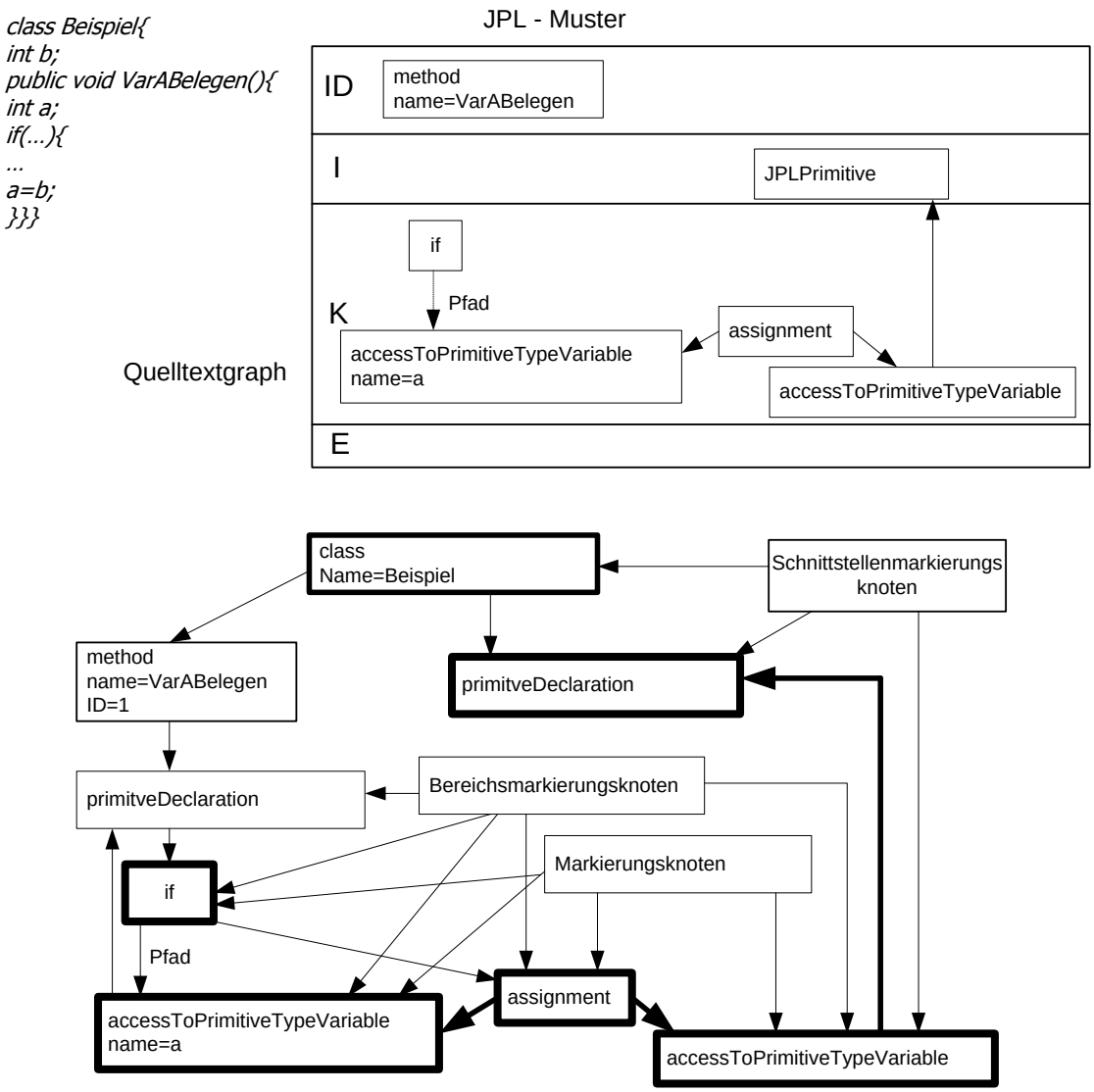


Abbildung 53: Graph eines Einzelmusters nach Ablauf des Algorithmus

Die Erstellung der Pfadkante (Schritt 6) wird weiter unten im Text separat behandelt. Im Folgenden wird das in der Körpersektion spezifizierte Muster gesucht und mit einem Markierungsknoten verbunden (Schritt 7). Da im Import des Suchmusters ein JPLPrimitive-Knoten spezifiziert wurde, werden nachfolgend die mit dem Match der Körperstruktur verbundenen Übergabestrukturen, in diesem Fall die Nutzung einer Membervariablen, markiert (Schritt 10). Da ein Muster gefunden werden konnte, welches allen Kriterien entspricht (dicker Rahmen), ist die Mustersuche hiermit abgeschlossen. Das mehrfache Auftreten eines Musters wird nicht betrachtet.

Umsetzung des JPL-Pfades durch Graphregeln
In diesem Abschnitt werden Graphregeln erläutert, die notwendig sind, um die Suche nach dem Element *Pfad* realisieren zu können. Sie detaillieren somit Schritt 6 des zuvor dargestellten Algorithmus.

Zur Erinnerung: Über einen Pfad kann eine Verbindung zwischen zwei JCG-Elementen im Kontrollfluss spezifiziert werden, ohne die zwischenliegenden Elemente betrachten zu müssen. Die Elemente dürfen nicht in der gleichen Anweisung liegen. Der Pfad wird innerhalb eines Gültigkeitsbereichs gesucht und geht nicht über diesen hinaus. Die Herausforderung, einen Pfad im Quellcodegraphen zu suchen, liegt darin, dass indirekte Beziehungen im JCG nicht spezifiziert sind. Da der vollständige JCG betrachtet werden muss, sind umfangreiche Hilfsstrukturen notwendig, um diese indirekten Beziehungen direkt darzustellen und suchen zu können. Um diese zu minimieren, setzt der im Weiteren gezeigte Algorithmus auf der transitiven Hülle des Kontrollabhängigkeitsgraphen auf, der bereits eine Hilfsstruktur für die notwendigen indirekten Beziehungen enthält.

Der im Folgenden beschriebene Algorithmus besitzt die Vorbedingung, dass ein JCG-Knoten Quelle oder Ziel nur eines Pfades im JPL-Muster sein darf. Mehrfachverbindungen eines Knotens durch *Pfad*-Kanten werden nicht betrachtet.

Die Regeln, welche die Suche nach dem Pfad realisieren, setzen auf der Struktur der transitiven Hülle des Kontrollabhängigkeitsgraphen auf, welche zuvor über dem/den Gültigkeitsbereich(en) des Quellcodegraphen erstellt wurde, der im Rahmen der Pfadsuche analysiert werden muss (Gültigkeitsbereich des Moduls, welches den/die Pfad(e) enthält). Die transitive Hülle der Kontrollabhängigkeiten wird aus folgenden Gründen für die Pfadsuche genutzt:

- Für die Pfaderkennung selbst ist kein Graphdurchlauf notwendig. Dieser erfolgt einmalig bei der Erstellung des AJSDG und der transitiven Hülle, so dass für die Erkennung der Pfade nur direkte Beziehungen über die transitiven Kontrollabhängigkeitskanten untersucht werden müssen.
- Gültigkeitsbereiche sind in der transitiven Hülle bereits berücksichtigt, d.h. es ist sichergestellt, dass Quell- und Zielknoten im Ablauf der Anweisungen aufeinander folgen, auch wenn diese zu unterschiedlichen Gültigkeitsbereichen gehören. Bei direkter Nutzung des JCG muss hierzu ein eigener Algorithmus generiert werden.

Für diesen Performancegewinn wird in Kauf genommen, dass die AJSDG-Struktur und die transitive Hülle über die Kontrollabhängigkeitsbeziehungen evtl. nur für die Pfadsuche erstellt werden müssen. Dieser Nachteil relativiert sich allerdings, da für diese Arbeit von der Annahme ausgegangen wird, dass die meisten JPL-Muster AJSDG-Elemente enthalten und der AJSDG-Graph bereits zur Suche nach diesen Elementen in die Quelltextrepräsentation eingefügt werden muss. Diese Annahme ergab sich aus Beobachtungen im Rahmen der Erstellung der Anwendungsbeispiele für Kapitel 8. Somit ist zur Durchführung der Pfadsuche zusätzlich nur noch die Generierung der transitiven Kontrollabhängigkeiten notwendig, so diese nicht bereits im Muster enthalten sind und diese Kanten bereits für deren Prüfung erstellt werden müssen. Aus Performancesicht ist weiterhin entscheidend, wie viele Pfade im Suchmuster spezifiziert werden. Je mehr Pfade im JP-Muster beschrieben werden, desto stärker können die Performancevorteile einer Suche über die transitive Hülle der

Kontrollabhängigkeiten gegenüber einer Suche, die direkt auf dem JCG aufsetzt, genutzt werden.

Hinweis: Aus den Ausführungen oben folgt, dass für Muster, welche keine AJSDG-Elemente und nur wenige Pfadstrukturen enthalten, aus Performancesicht der direkte Durchlauf über den JCG Performancevorteile hat. Der entsprechende Algorithmus durchläuft den Graphen des zu analysierenden Gültigkeitsbereichs, beginnend an dessen Wurzel über die Kanten vom Typ *consecutive*, welche die Java-Ausdrücke, welche selber Teilbäume bilden, miteinander verbinden.

Für diese Arbeit wird allerdings davon ausgegangen, dass JPL-Muster meistens Elemente des AJSDG enthalten, und/oder bei der Nutzung von Pfadstrukturen diese Mehrfach verwendet wird. Somit wird im Folgenden die Pfaderstellung über die transitive Hülle der Kontrollabhängigkeiten beschrieben.

Dem Algorithmus der Pfadsuche liegt folgende Erkenntnis zugrunde:

Wenn AJSDG-Knoten zweier Ausdrücke über eine transitive Kontrollabhängigkeitskante miteinander verbunden sind, dann ist die dem Quellknoten zugeordnete JCG Struktur der JCG-Struktur des Zielknotens vorgelagert. Dies bedeutet, dass der Ausdruck (und damit auch alle Teile des Ausdrucks) des Quellknotens im Quelltext vor dem Ausdruck des Zielknotens liegt.

Die Erkenntnis folgt aus der Struktur des AJSDG, in welchem alle Quellknoten transitiver Kontrollabhängigkeitskanten in einer höheren Hierarchieebene liegen als die Zielknoten. Die Hierarchieebenen beziehen sich hier auf die Gültigkeitsbereiche von Variablen.

Um auch die Reihenfolge von Anweisungen auf einer Hierarchieebene erfassen zu können ist es notwendig, auf das Attribut der Kontrollabhängigkeitskante, welches den Platz der Anweisung des Zielknotens in der Anweisungsreihenfolge im Quelltext repräsentiert, zu referenzieren. Es dürfen nur Knoten betrachtet werden, welche in der Sequenz nach der Anweisung des Quellknotens liegen (s. Regel in

Abbildung **56**).

Die folgende Regelsequenz realisiert die Suche nach einem Pfad, basierend auf der gegebenen Struktur der transitiven Hülle. Die Knoten, welche Quelle und Ziel des Pfades repräsentieren, sind entsprechend der Angaben im konkreten Suchmuster zu parametrisieren, um die Regeln im Rahmen eines Suchablaufs anwenden zu können. Die Regeln werden im Rahmen der Vorbereitung des Quelltextgraphen auf die Mustersuche (Erstellung von Hilfsstrukturen) eingesetzt (s. Algorithmen zur Mustersuche in den Kapiteln 5.9. und 5.10.).

<u>Regelsequenz zur Pfadsuche:</u>

1. Suche den AJSDG-Knoten, der einem möglichen JCG-Ursprungselement des Pfades (Quellknoten der Pfadkante des Musters) in der Quellcoderepräsentation zugeordnet ist. Markiere den/die AJSDG-Knoten mit dem Knotentyp *source* und den Knoten, von dem dieser kontrollabhängig ist mit dem Knotentyp *sourcesearch*. Die möglichen JCG-Quellelemente im Gültigkeitsbereich wurden bereits im Rahmen der Bereichsmarkierung im Quellcodegraphen markiert, s. Prozesspunkt 2 im Grobalgorithmus. Falls kein Knoten gefunden werden kann,

bricht der Algorithmus mit einer Fehlermeldung ab. Da kein Pfad gefunden werden kann, ist es auch nicht möglich das Suchmuster, welches den Pfad enthält, im Quellcode zu finden. Es wird ein Ergebnisknoten mit der entsprechenden Fehlermeldung erstellt.

Hinweis: Die ID eines JCG-Knotens, welche diesen mit einem AJSDG-Knoten in Beziehung setzt, wird über einen Hilfsknoten realisiert. Zur besseren Übersichtlichkeit wird diese im Folgenden direkt im JCG-Knoten angegeben.

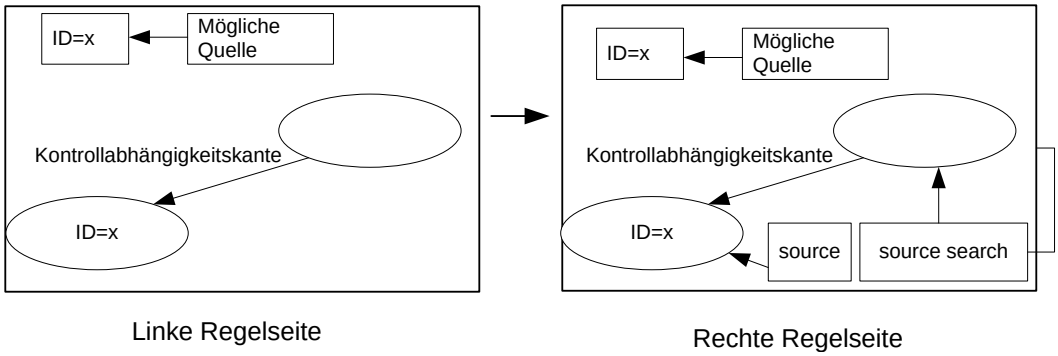


Abbildung 54: Durchführung der Suche nach dem Pfad Element 1

2. Markiere alle AJSDG-Knoten, welche einem möglichen JCG-Zielelement des Pfades zugeordnet sind, mit dem Knotentyp *targetsearch*. Die möglichen JCG-Zielelemente im Gültigkeitsbereich wurden bereits im Rahmen der Bereichsmarkierung im Quellcodegraphen markiert, s. Prozesspunkt 2 im Grobalgorithmus. Falls kein Knoten gefunden werden kann, bricht der Algorithmus mit einer Fehlermeldung ab.

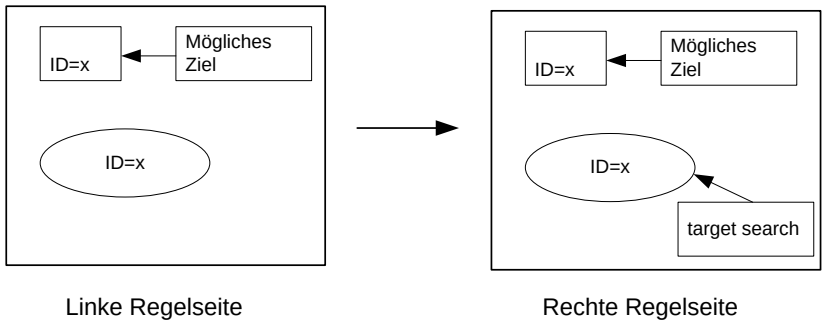


Abbildung 55: Durchführung der Suche nach dem Pfad Element 2

3. Suche eine Kontrollabhängigkeitskante, welche einen mit *sourcesearch* und einen mit *source* markierten Knoten miteinander verbindet, und markiere die Knoten mit *source*, deren Ausdruck sequentiell hinter dem bereits markierten Ausdruck liegen.

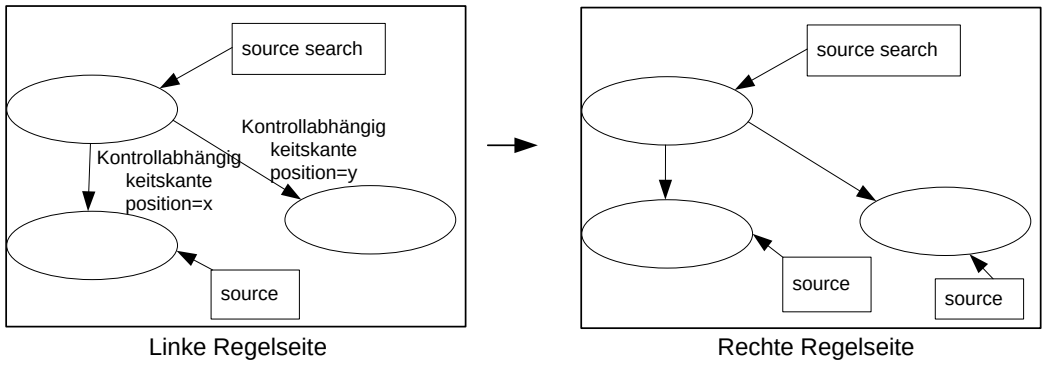


Abbildung 56: Durchführung der Suche nach dem Pfad Element 3

Hinweis: Durch eine Condition werden Attribute zweier oder mehrerer Knoten oder Kanten der linken Seite einer Regel miteinander in Beziehung gesetzt. Falls die Bedingung nicht erfüllt ist, wird die Transformation nicht ausgeführt.

4. Suche eine Kante der transitiven Hülle über den Kontrollfluss, welche einen mit s*ource* und einen mit *targetsearch* markierten Knoten miteinander verbindet und verbinde die beiden Knoten.

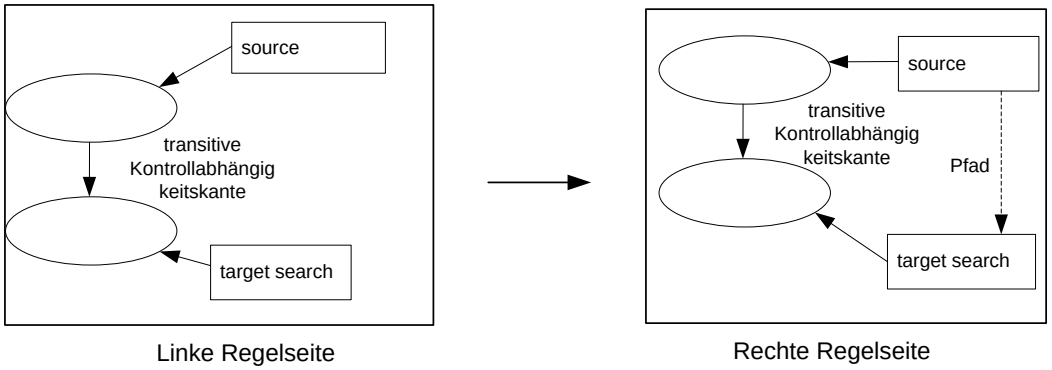
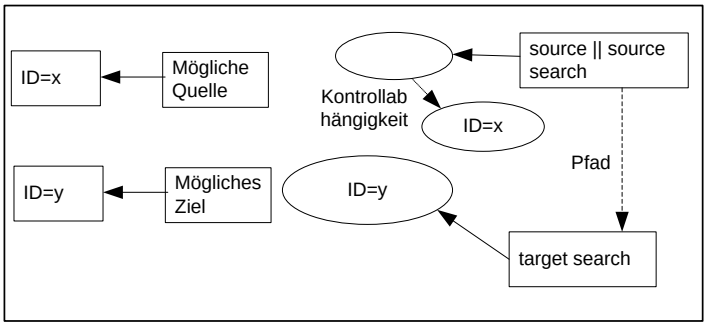


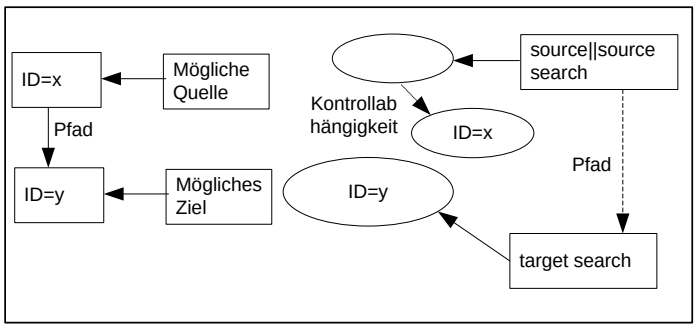
Abbildung 57: Durchführung der Suche nach dem Pfad Element 4

Durch diesen Algorithmus werden alle möglichen Quellelemente mit allen möglichen Zielelementen im AJSDG verbunden.

5. Nachfolgend werden die Pfade in die JCG-Struktur des Quelltextes übertragen, so dass die JCG Struktur im Modulkörper direkt auf dem Graph des Quelltextes gesucht werden kann.



Linke Regelseite



Rechte Regelseite

Abbildung 58: Durchführung der Suche nach dem Pfad Element 5

Bedingungen des Regelsatzes:

Vorbedingung: Es existieren JCG-Knoten, die auf Grund ihres Knotentyps als mögliche Quelle und mögliches Ziel eines Pfades identifiziert wurden.

Nachbedingung: Zwischen den zuvor identifizierten JCG-Knoten, die durch eine transitive Kontrollabhängigkeit miteinander verbunden sind, werden Kanten vom Typ *Pfad* erstellt.

Im Folgenden wird die Pfadsuche an einem Beispiel verdeutlicht, welches den Quellcode aus dem Beispiel in Abbildung 52 erweitert.

Die Abbildung unten zeigt die Ergebnisstruktur nach Durchführung der Pfadsuche, welche im JPL-Muster aus Abbildung 52 spezifiziert wurde. Der Quelltext wurde um eine while-Schleife, welche in die if-Bedingung eingebettet ist, erweitert. Hierdurch soll verdeutlicht werden, dass ein Pfad auch in eingebetteten Strukturen gefunden wird. Das dargestellte JPL-Muster betrachtet nur die Körpersektion des Moduls.

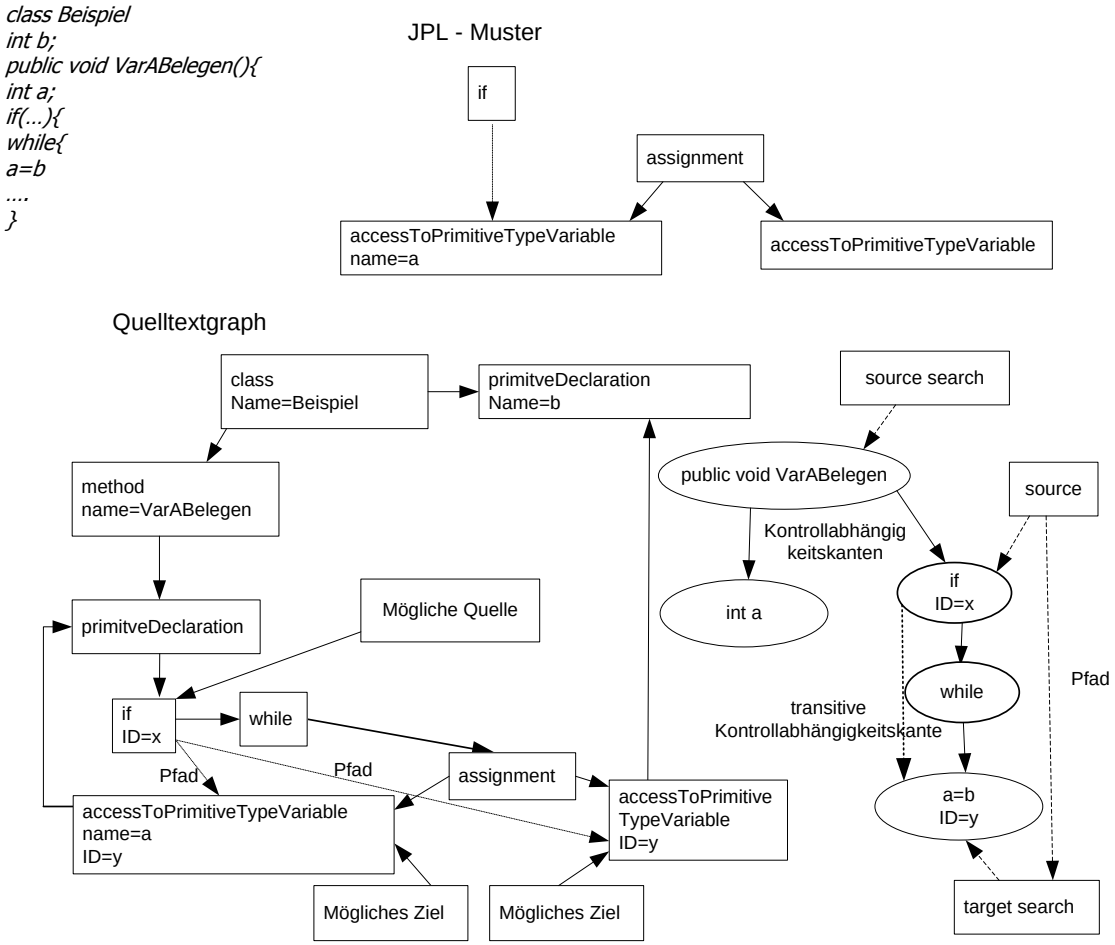


Abbildung 59: Beispiel für die Suche nach einem Pfad

Der unter dem Muster abgebildete Quelltextgraph zeigt die Situation nach erfolgreicher Mustersuche. Zuerst werden die möglichen Quell- und Zielknoten im Quelltextgraph markiert (Prozesspunkt 2 im Grobalgorithmus). Danach werden die zugeordneten AJSDG-Knoten markiert (Schritt 1 und 2 in der Pfadsuche). Nachfolgend wird über die Regeln in Schritt 3 und 4 nach Kontrollabhängigkeiten und transitiven Kontrollabhängigkeiten gesucht. Für den Match (die gefundene Kante ist bezeichnet mit *transitive Kontrollabhängigkeitskante*) wird der Pfad sowohl in die AJSDG-Struktur als auch in die JCG Struktur eingetragen. Zwei mögliche Pfade wurden identifiziert.

Nachdem in diesem Kapitel der Prozess der Ausführung der Mustersuche für ein einzelnes Modul dargestellt wurde, wird im folgenden Kapitel die Mustersuche über ein komplexes JPL-Muster betrachtet.

5.10. Sektionsbasierte Mustersuche für JPL-Muster

Für die Ausführung der sektionsbasierten Mustersuche über komplexe Suchmuster, die mehrere miteinander verbundene Module enthalten, wird sowohl der im letzten Kapitel beschriebene Prozess der Mustererkennung für einzelne JPL-Module verwendet, als auch ein Prozess, welcher die Muster, bzw. die gefundenen Matches, der einzelnen Module miteinander kombiniert. Dieser Vorgang wird im Folgenden erläutert.

Bei der Durchführung der Mustersuche von zwei oder mehr miteinander verbundenen Modulen müssen mehrere Teilmuster kombiniert werden:

1. Verschiedene mögliche Gültigkeitsbereiche (G1 - Gx) pro Modul.
2. Die verschiedenen möglichen Matches der Körpersektion in den Gültigkeitsbereichen (K1 - Kx) pro Modul.
3. Die verschiedenen Variablenübergabestrukturen (V1 – Vx) zwischen den Gültigkeitsbereichen.

Folgende Prozessschritte realisieren die Identifikation eines Gesamtmatches, basierend auf den Ergebnissen der Matches zu Gültigkeitsbereichen, Mustern in der Körpersektion und Übergabestrukturen. Hierbei wird der Prozess der Mustersuche für ein einzelnes Muster (s. Kapitel 5.9.) um Schritte ergänzt, die Verbindungen mehrerer Module untereinander mit einbeziehen.

1. Alle möglichen Matches der Körper-Sektionen der JPL-Module des komplexen Suchmusters werden durch Ausführung des Prozesses in Kapitel 5.9. (Schritte 1 bis 8 in der Einzelmustersuche) gesucht und die Knoten jedes Matches werden mit einem Markierungsknoten verbunden. Die Regeln zur Bereichserkennung werden um Strukturen für die hierarchische Kante (die Knoten des hierarchisch tieferen Bereichs müssen zuvor mit denen des höher stehenden Bereichs markiert worden sein) und für semantisch zusammenhängende Blöcke (die Strukturen der verbundenen Identifikatorsektionen werden gemeinsam in einem Muster gesucht) ergänzt. Wird nicht für jedes Modul mindestens ein Match gefunden, so bricht der Prozess mit der Erstellung eines Ergebnisknotens ab, der besagt, dass das Muster nicht gefunden wurde.
2. Nun werden die Beziehungen zwischen den Körpersektionen betrachtet. Unter Verwendung der Regeln in den Kapiteln 5.4., 5.5. und 5.6. wurden die JPL-Schnittstellenknoten abgeleitet und die im Folgenden zu suchenden Übergabevarianten erstellt. Nachdem in Schritt eins alle Matches für alle Körpersektionen identifiziert und markiert wurden, werden alle möglichen Matches für die Variablenübergabestrukturen gesucht (Schritte 9 bis 11 im Algorithmus für die Einzelmustersuche). Sobald ein Übergabemuster gefunden wird, werden die Knoten der Übergabestruktur mit einem Schnittstellenknoten markiert (s. Kapitel 5.7.).
3. Nachfolgend wird analysiert, ob ein Gesamtmuster identifiziert werden kann (zu Details s. Kapitel 5.7., Schritt 7), welches alle Schnittstellenmarkierungen und Markierungen der Körpersektion in einem Suchmuster vereinigt. Das Ziel dieser auf einzelnen Markierungsknoten basierenden Struktur liegt in der Identifikation eines Gesamtmusters, welches die zuvor identifizierten und markierten Teilgraphen

erfasst. Dieser Prozesspunkt erweitert den Schritt 11 im Ablauf der Suche zu einem Einzelmuster.

4. Falls das in Punkt drei beschriebene Muster gefunden wird, so wurde die komplexe Struktur erkannt und ein Ergebnisknoten wird generiert.

Das folgende Beispiel erweitert die in Kapitel 5.9. beschriebene Aufgabe um die Abfrage einer Anforderung, welche sich über zwei Gültigkeitsbereiche erstreckt. Hierdurch wird die Anwendung der oben dargestellten Prozessschritte für komplexe Muster demonstriert.

Das JPL-Muster sucht im Quelltext eine Struktur, die folgende Anforderung erfüllt: In der Methode *VarABelegen* wird die Variable *a* belegt. Der Wert einer Variablen, welche in der Methode *setAlter* belegt wird, wird der Variablen *a* zugewiesen.

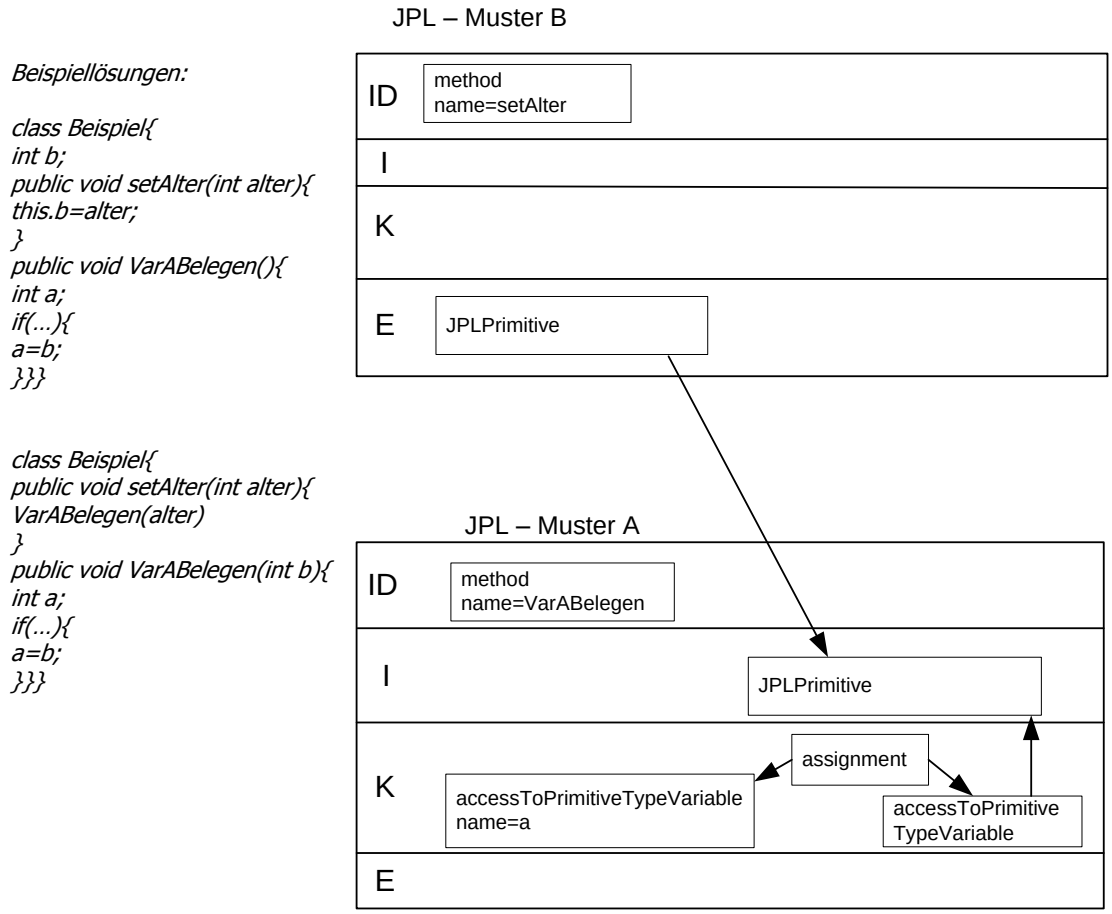


Abbildung 60: Beispiel zur Suche nach komplexen JPL-Mustern 1

Auf der linken Seite sind zwei Beispielquelltexte dargestellt, an denen im Folgenden die Durchführung der Mustersuche demonstriert wird. Die Beispiellösungen wurden ausgewählt, da sie unterschiedliche Typen der indirekten Variablenübergabe beinhalten. Während in Beispiel eins die Membervariable *b* zur Variablenübergabe verwendet wird, erhält in Beispiel zwei der Parameter *b* den zu übergebenden Wert.

Prozess der Mustersuche zu Quelltext 1:

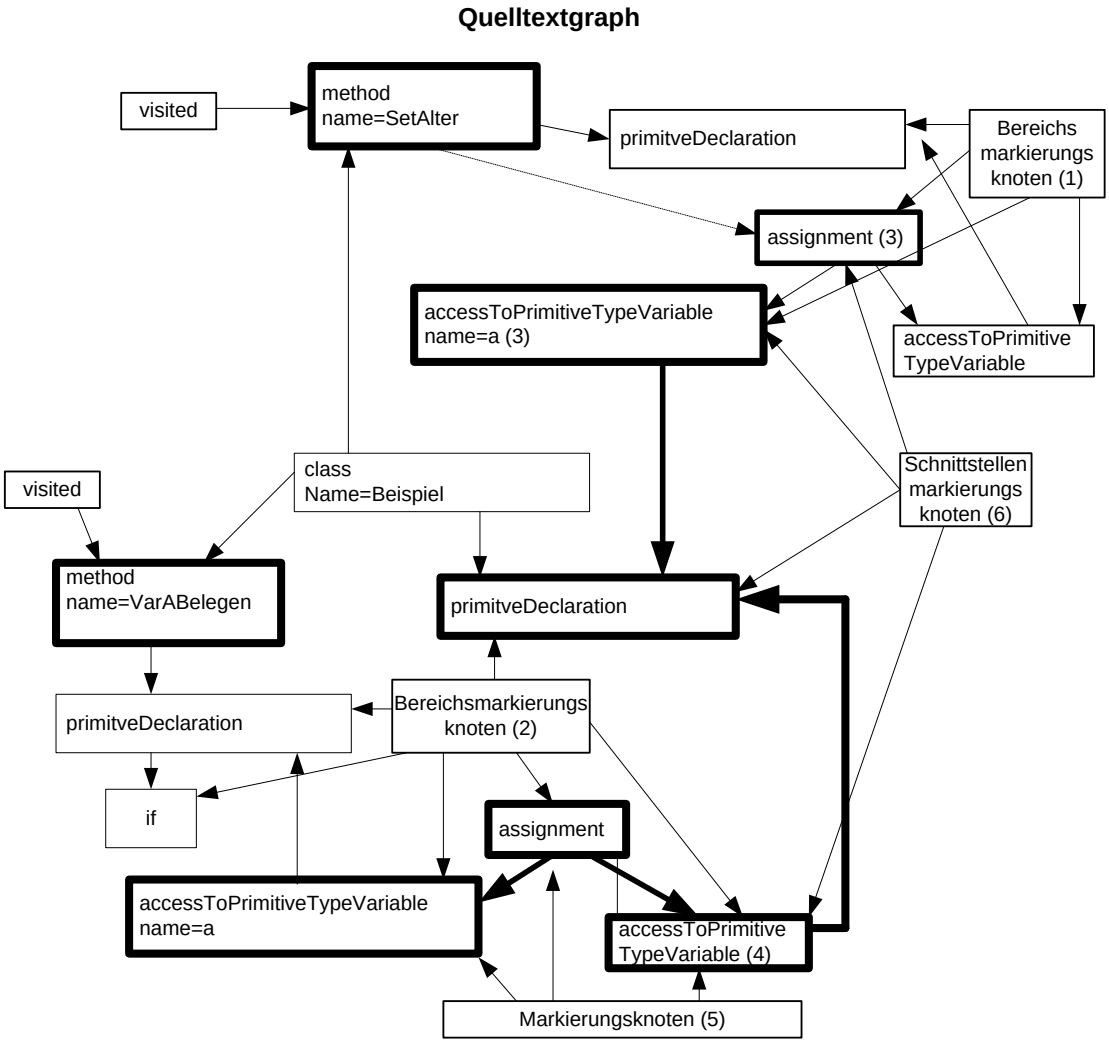


Abbildung 61: Beispiel zur Suche nach komplexen JPL-Mustern 2

Obige Abbildung zeigt die Graphrepräsentation des ersten Beispielquelltextes. Im ersten Schritt werden die beiden Muster einzeln im Quelltext gesucht. Das Muster zur Methode *setAlter* enthält in der Körpersektion keine Elemente, so dass hier nur der Methodenknoten gefunden und der entsprechende Gültigkeitsbereich markiert wird. In der Methode *VarABelegen* ist in der Körpersektion eine Zuweisungsstruktur zweier *accessToPrimitiveType*-Referenzen definiert, welche entsprechend zusätzlich zum Bereich markiert werden. Damit ist Schritt 1 abgeschlossen.

In Schritt 2 werden Muster für Implementierungsvarianten bezogen auf eine Übergabe zwischen den Methoden *setAlter* und *VarABelegen* gesucht. Im Beispiel wird durch die nachfolgende Mustersuche die mit dem Schnittstellenmarkierungsknoten markierte Struktur gefunden. Die entsprechende Implementierungsvariante wird in Kapitel 5.6. beschrieben*: Werteübergabe durch Nutzung einer Variablen in einem hierarchisch übergeordnetem Gültigkeitsbereich*.

In Schritt 3 werden die in Schritt 1 und Schritt 2 gefundenen Muster durch ein Gesamtmuster erfasst.

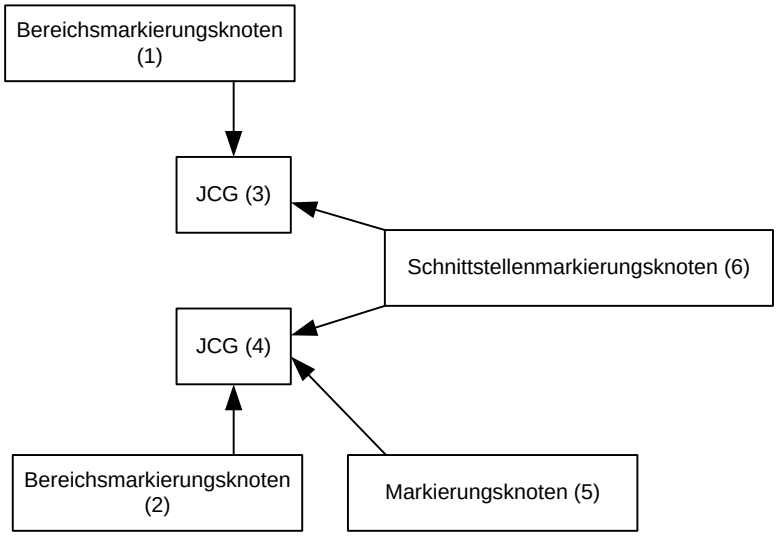


Abbildung 62: Muster zur Identifikation des Gesamtmusters

In Abbildung 61 sind die Knoten, die einen Match zum Muster oben erzeugen, mit korrespondierenden in Klammern gesetzten Nummern beschriftet. Durch die erfolgreiche Suche nach dem Gesamtmuster wurde das Muster gefunden und ein Ergebnisknoten wird erstellt.

Prozess der Mustersuche zu Quelltext 2:

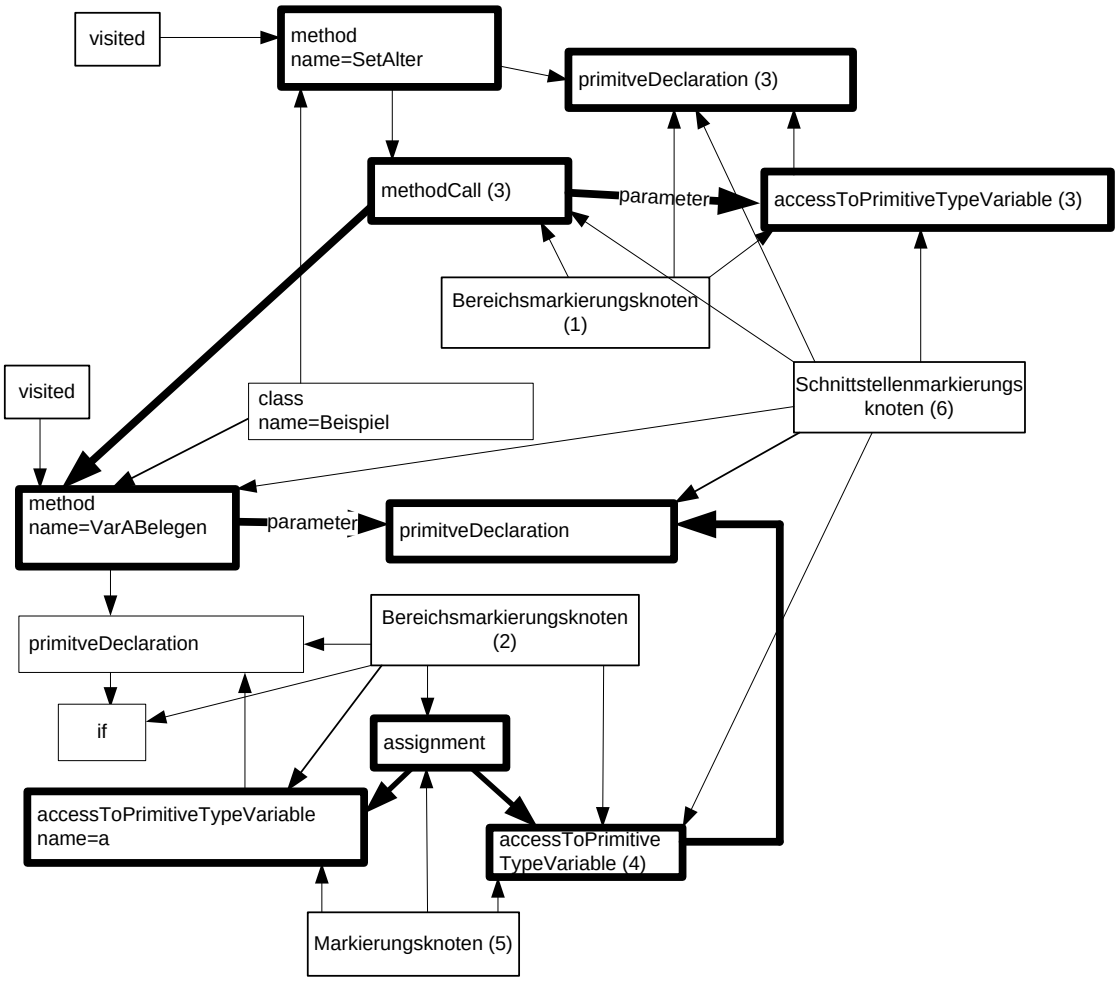


Abbildung 63: Beispiel zur Suche nach komplexen JPL-Mustern 3

Obige Abbildung zeigt die Graphrepräsentation des zweiten Beispielquelltextes. Der erste Prozessschritt erfolgt äquivalent zum vorhergehenden Beispiel. In der Abbildung sind die gefundenen Elemente auch hier mit den entsprechenden Markierungsknoten verbunden.

In Schritt 2 werden Matches für Implementierungsvarianten bezogen auf eine Übergabe zwischen den Methoden *setAlter* und *VarABelegen* gesucht. Im Beispiel wird durch die nachfolgende Mustersuche die mit den Schnittstellenmarkierungsknoten markierte Struktur gefunden. Die entsprechende Implementierungsvariante wird in Kapitel 5.6. beschrieben: *Verbindungstyp: Objektvariable wird als Parameter an eine Methode übergeben*. Die Variante wurde hier für einen.

In Schritt 3 werden die in Schritt 1 und Schritt 2 gefundenen Muster zu einem Gesamtmuster zusammengeführt und die Suche nach dem Gesamtmuster durchgeführt. Das Muster ist in Abbildung 62 dargestellt. In der Abbildung oben sind die Knoten, die einen Match zum Muster in Abbildung 62 erzeugen, mit korrespondierenden in Klammern gesetzten Nummern beschriftet. Zur Identifizierung des Knotens mit der Nummer 3 gibt es drei Möglichkeiten, d.h. drei valide Matches. Damit wurde das Muster gefunden und ein Ergebnisknoten wird erstellt.

Nach der Beschreibung der Suchmusterspezifikation und hierauf folgend der Durchführung der Suche auf der Graphrepräsentation des Quelltextes, wird im Folgenden die Technik des Slicings auf der transitiven Hülle dargestellt, welche für verschiedene Methoden zur Entwicklung von JPL-Spezifikationen benötigt wird.

5.11. Slicing

Um die in Kapitel 6.3. beschriebene Methodik der Bottom-Up-Analyse über Abhängigkeitsbeziehungen zu unterstützen, wird in diesem Kapitel die Technik des Slicings vorgestellt. Diese Methodik geht davon aus, dass über die Betrachtung typischer Variablenabhängigkeiten (Kontrollabhängigkeiten und Datenabhängigkeiten) in gegebenen Übungslösungen, Suchmuster mit geringerem Aufwand erstellt werden können, als dies bei Betrachtung des vollständigen Quelltextes möglich ist. Hierbei wird angenommen, dass über die Suche nach für die Musterlösung typischen Abhängigkeiten, bereits eine Vielzahl möglicher Lösungsvarianten erfasst werden können. Hierdurch ist es für den Übungsleiter nicht mehr notwendig, den vollständigen Quelltext der Übungslösungen zu analysieren, sondern er kann sich auf Quelltextfragmente beschränken, welche Variablenabhängigkeiten, z.B. ausgehend von einer festgelegten Variablen, abbilden (z.B. die separate Darstellung der Quelltextfragmente und Abhängigkeitsbeziehungen, die sich auf einen Parameter einer Methode beziehen, und nachfolgende Übernahme der Kontroll- und Datenabhängigkeitsbeziehungen, in das zu erstellende Suchmuster).

Die Technik des Slicings wurde von Marc Weiser entwickelt, welcher in seinen Arbeiten die Extrahierung eines Programmfragments basierend auf Variablenabhängigkeiten untersuchte und in [Wei81] einen Algorithmus vorstellte, der dieses Codefragement, basierend auf einem AST-Graphen, automatisch identifizieren kann. Es wird zwischen Forward- und Backward-Slices unterschieden. Bei einem Forward-Slice wird, ausgehend von einem festgelegten Ausdruck im Quellcode, das Codefragment ermittelt, welches von diesem Ausdruck datenabhängig ist. Ein Backward-Slice hingegen beschreibt das Codefragment, von dem ein zuvor festgelegter Ausdruck im Quelltext datenabhängig ist.

In [OO84] haben Ottstein und Ottstein einen Slicingalgorithmus entwickelt, welcher auf der Struktur des Programmabhängigkeitsgraphen aufsetzt. Dieser Algorithmus ist wesentlich performanter als der von Weiser vorgestellte, da durch die vorherige Erstellung des Programmabhängigkeitsgraphen grundlegende Codeanalysen vorweggenommen wurden und somit während des Slicingvorganges nicht mehr notwendig sind. Interprozedurales Slicing wurde nachfolgend in [HRB90] beschrieben, wobei der Algorithmus von Ottstein auf den Systemgraph ausgeweitet wurde. Auf diesem Ansatz basiert der in diesem Abschnitt dargestellte Vorschlag zum Slicing durch Graphtransformation, welcher in Kapitel 6.4. im Rahmen der „Bottom-Up-Strategie mittels Variablenabhängigkeiten“ eingesetzt wird.

Im Folgenden wird eine Regelfolge gezeigt, welche einen Slicing-Algorithmus zum Forward-Slicing über Graphregeln auf dem AJSDG realisiert. Der Algorithmus basiert auf dem SPO-Ansatz, so dass bei der Entfernung von Knoten automatisch alle Kanten, mit denen diese verbunden waren, auch entfernt werden.

Ausgehend vom Startknoten (*Init*) wird die transitive Hülle zum Datenabhängigkeitsgraphen mit Slicingkanten dekoriert. Nachfolgend werden die nicht zum Slice gehörenden AJSDG- und JCG-Knoten und Kanten gelöscht.

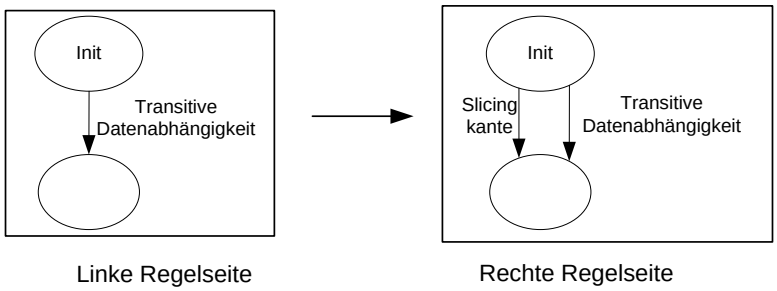


Abbildung 64: Regel 1 zur Generierung eines Forward-Slices

Löschen der vom Slice nicht betrachteten AJSDG-Knoten:

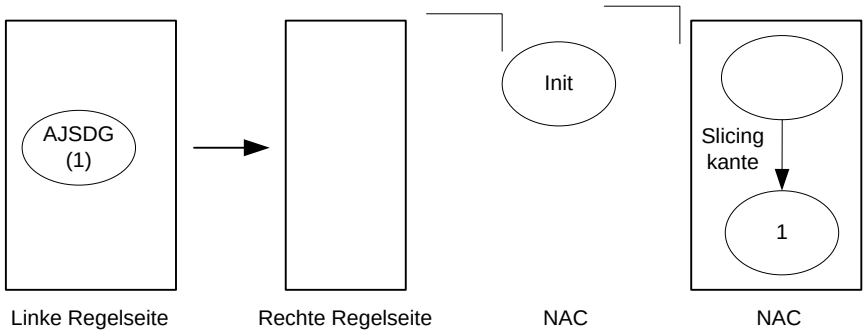


Abbildung 65: Regel 2 zur Generierung eines Forward-Slices

Löschen der JCG-Knoten, welche nicht mit AJSDG-Knoten verbunden sind:

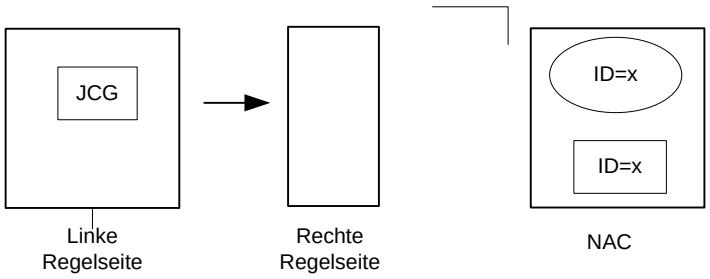


Abbildung 66: Regel 3 zur Generierung eines Forward-Slices

Nach Ausführung dieser Regelfolge bleibt lediglich der Codefragmentgraph zurück, welcher von dem initial festgelegten Ausdruck (initialer AJSDG-Knoten) datenabhängig ist.

Neben der direkten Unterstützung der in Kapitel 6.3. „Bottom-Up über Variablenabhängigkeitsbetrachtung“ vorgestellten Vorgehensweise zur Mustererstellung, kann die Technik des Slicings auch für die in Kapitel 6.2. „Bottom-Up über Syntaxanalyse“ und Kapitel 6.4. „Bottom-Up mittels eines gegebenen Musterkataloges“ beschriebenen Vorgehensweisen eingesetzt werden, um die Analyse der gegebenen Musterlösungen zu vereinfachen

5.12. Zusammenfassung

In Kapitel 5 wurden, nach einer Einführung in die für dieses Kapitel benötigten Grundlagen der Graphtransformation, zwei Varianten vorgestellt, nach denen die Mustersuche durchgeführt werden kann. Zum einen kann diese über die Suche nach Gesamtmustern zu den Implementierungsvarianten realisiert werden, und zum anderen kann das Muster in seine einzelnen Module und deren Sektionen unterteilt und der Quellcodegraph nach Teilmustern durchsucht werden, die abschließend für einen Gesamtmatch kombiniert betrachtet werden müssen. Die Vorteile und Nachteile der beiden Alternativen wurden aufgeführt, so dass der Nutzer hierdurch eine Entscheidungsgrundlage erhält, welche Variante im bei ihm gegebenen Kontext zu wählen ist.

Nachfolgend wurde für beide Alternativen die Ableitung der abstrakten modularisierten JPL-Suchmuster in eine Menge von „flachen“, nicht mehr modularisierten, Suchmustervarianten, welche nur noch JCG- und AJSDG-Elemente enthalten, dargestellt. Im Fokus standen die abstrakten JPL-Schnittstellenknoten, über deren Auflösung Suchmuster für die verschiedenen Varianten der Werteübergabe zwischen Gültigkeitsbereichen erstellt werden. Die Ableitung der Suchmuster erfolgte über die Technik der Graphtransformation, so dass die Transformationsschritte visuell spezifiziert werden können, ohne einen Modellbruch von der visuellen Spezifikation in JPL zu den Suchmustervarianten durchführen zu müssen.

Weiterhin wurde die automatisierte Erstellung der Transitiven Hülle über dem Quellcodegraphen beschrieben. Die hier erzeugten Elemente ergänzen die in Kapitel 3 aufgeführten Sorten zur Quelltextdarstellung, so dass nun die Erstellung aller Strukturen erläutert wurde, auf denen im folgenden die Mustersuche durchgeführt wird. Die Darstellung wurde in diesem Kapitel platziert, da die Erstellung der Transitiven Hülle über die Technik der Graphtransformation erfolgt, die in Kapitel 5 eingeführt wurde.

Anschließend wurde gezeigt, wie der Prozess der Mustersuche für beide Varianten der Mustersuche, die zu Anfang dieses Kapitels angesprochen wurden, realisiert wird. Ausgehend von der Suche nach einzelnen Mustern, wurde nachfolgend die Suche nach komplexen Mustern, welche aus mehreren Modulen bestehen, dargestellt und der Ablauf der Suche über ein Beispiel verdeutlicht. Im Fokus stand hier die sektionsbasierte Suche, in welcher die einzelnen Bereiche des modularisierten Gesamtmusters getrennt voneinander betrachtet werden. Die Matches dieser einzelnen Sektionen werden nachfolgend miteinander kombiniert, um das Gesamtmuster zu identifizieren.

Abschließend wurde die Technik des Slicings und deren Anwendung auf die Quelltextrepräsentation dargestellt, um die nachfolgend beschriebene Methode der Bottom-Up-Analyse zu unterstützen.

Im folgenden Kapitel werden verschiedene Strategien vorgestellt, um JPL-Spezifikationen zu gegebenen Anforderungen konsistent zu erstellen, und somit die JPL effektiv anwenden zu können.

6. Vorgehensweisen für die Spezifikation von JPL-Mustern

Zur Erstellung von Suchmustern in der JPL können unterschiedliche Vorgehensweisen angewendet werden. In diesem Kapitel werden vier verschiedene Prozesse vorgestellt, nach denen Übungsleiter JPL-Spezifikationen zu ihren Übungsaufgaben erstellen können. Die Kriterien zur Prozessauswahl bilden zum einen der Kontext, in dem der Quelltext, welcher nachfolgend geprüft werden soll, erstellt wird, und zum anderen der Kontext, in dem die Suchmuster erstellt werden.

Es wird unterschieden zwischen:

1. Top-Down ausgehend von funktionalen Anforderungen: In diesem Prozess werden die Suchmuster ausgehend von den Anforderungen der Übungsaufgabe erstellt.
2. Bottom-Up über Syntaxanalyse: Dieser Prozess basiert auf einem bereits vorliegendem Quellcode (z.B. alte Übungsaufgaben, oder bereits erstellte Musterlösungen), über den auf die zu suchenden Muster geschlossen wird.
3. Bottom-Up über Variablenabhängigkeiten: In diesem Prozess wird ausgehend von Variablenabhängigkeiten, die im zu untersuchenden Code enthalten sein müssen, auf Suchmuster geschlossen.
4. Bottom-Up mittels Patternkatalog: Hier wird ein bereits erstellter Suchmusterkatalog zur Unterstützung bei der Erstellung neuer Muster hinzugezogen.

In jeder dieser Vorgehensweisen muss der Nutzer entscheiden, welche Sprachelemente der JPL er im konkreten Fall einsetzt und miteinander kombiniert. Dies bedeutet, dass er den Detaillierungsgrad des Suchmusters festlegt, und somit über die Höhe des Risikos des Auftretens von *false negatives* und *false positives* entscheidet. Zu beachten ist hierbei, dass von dieser Festlegung die Anzahl der zu erstellenden Suchmustervarianten abhängt. Weitere Überlegungen zu Vorteilen und Nachteilen abstrakter bzw. detaillierter Suchmuster sind zu Anfang des Kapitels 2.7 zu finden.

Folgende Abstraktionsebenen zur Quelltextanalyse werden unterschieden:

1. Analyse der Codestruktur auf Klassenstrukturebene. Betrachtung von Klassenhierarchien, Interfaces und Methodenköpfen.
2. Analyse über die Kopplung von Methoden, bzw. Klassen (interne und externe Methodenaufrufe), sowie hiermit gekoppelt die Analyse der entsprechenden Datenflüsse.
3. Analyse der JCG- und AJSDG-Strukturen innerhalb einer Methode.

In den Mustern der verschiedenen Abstraktionsebenen können folgende JPL-Sprachelemente verwendet werden:

1. Ausschließliche Verwendung der Elemente des JCG
2. Ausschließliche Verwendung der Elemente des AJSDG
3. Kombinierte Verwendung von Elementen des JCG, des AJSDG sowie der Schnittstellenknoten

Die Verwendung von modularisierten komplexen Suchmustern erlaubt es Einzelmuster mit unterschiedlichen Detaillierungsgraden zu einem komplexen Suchmuster zusammenzufügen.

6.1. Top-Down ausgehend von funktionalen Anforderungen

Diese Vorgehensweise basiert auf den funktionalen Anforderungen der Übungsaufgabe und umfasst sowohl deren Analyse als auch die Ableitung von Suchmustern aus diesen. Die Anwendung dieses Prozesses bietet sich zur Erstellung von JPL-Musterspezifikationen an, deren Basis ausschließlich die gegebenen Anforderungen bilden.

Der Prozess gliedert sich in drei Stufen, basierend auf dem allgemeinen Vorgehen in der Anforderungsanalyse:

1. Klassifikation: Untersuchung der einzelnen Anforderungselemente und Entscheidung, welche Elemente durch ein Suchmuster spezifiziert werden.
2. Konstruktion: Erstellung der Suchmuster, wobei initial die einzelnen Module spezifiziert werden und diese nachfolgend zu einem komplexen Muster kombiniert werden.
3. Ablauf: Falls mehrere Suchmuster erstellt wurden, kann hier die Reihenfolge ihres Aufrufs festgelegt werden. Abhängigkeiten zwischen Mustern werden in diesem Schritt modelliert. Beispiel für eine Bedingung: Falls Muster a gefunden wurde, wird nicht nach Muster b gesucht.

Stufe 1a: Klassifikationstypen, abgeleitet aus den allgemeinen Typen der Anforderungsanalyse:

- Vollständige detaillierte Klassifikation: Abdeckung aller Lösungsmöglichkeiten auf Detailebene. Hierfür muss der strukturell vollständige JCG jeder möglichen Lösung zu einer Anforderung beschrieben werden. Hinweis: Strukturelle Vollständigkeit beinhaltet nicht die Abdeckung der Wertebereiche der Attribute von Knoten oder Kanten der JPL.
  - Vorteil: Abdeckung aller strukturellen Lösungsmöglichkeiten.
  - Nachteil: Auf Grund der großen Menge von Lösungsmöglichkeiten bei bereits sehr eingeschränkten Aufgabenstellungen bzw. Anforderungen, erfordert diese Vorgehensweise einen sehr hohen zeitlichen Aufwand.

- Vollständige abstrakte Klassifikation: Abdeckung aller Lösungsmöglichkeiten zu einer Anforderung, wobei nicht alle Lösungen vollständig detailliert beschrieben werden.
  - Vorteil: Im Vergleich zur „vollständigen detaillierten Klassifikation“ geringerer zeitlicher Aufwand der Suchmusterimplementierung.
  - Nachteil: Bei zu hoher Abstraktion ergibt sich das Problem der *false positives*.

- Unvollständige Klassifikation: Lediglich ein Teil der Anforderung und nur einige mögliche Lösungsstrukturen werden überprüft. Die Auswahl der zu prüfenden Elemente basiert auf Erfahrungen mit ähnlichen Übungsaufgaben:
    - Berücksichtigung von Erfahrungswerten, welche Strukturen am wahrscheinlichsten sind und Erstellung entsprechender Suchmuster. Nachteil: Implementierungen, welche Lösungsvarianten mit geringer Häufigkeit beinhalten, werden nicht spezifiziert und somit im Test als falsch deklariert (*false negatives*).
    - Einbezug von Erfahrungswerten, bei welchen Anforderungen viele Implementierungsvarianten möglich sind und Erstellung entsprechender Abstraktionen. Nachteil: Abstraktionen an diesen Stellen beinhalten die Gefahr der Erzeugung von *false positives*.
  - Vorteil: Geringerer zeitlicher Aufwand für die Musterspezifikation gegenüber der vollständigen Klassifikation.
  - Nachteil: Es können sich sowohl false positives, als auch false negatives ergeben.

Stufe 1b: Ableitung: Die weitere Spezifikation der Suchmuster basiert auf den Anforderungen, deren Realisierung im Code identifiziert werden soll, und dem gewählten Klassifikationstyp. Die Auswahl der zu erfassenden Anforderungen liegt in der Verantwortung des Übungsleiters und basiert auf deren Relevanz zur Bewertung der Lösung (Anforderungspriorisierung) und dem geschätzten zeitlichen Aufwand, welchen die Spezifikation der entsprechenden Suchmuster erfordert. Für die Ableitung der Suchmuster sind folgende Schritte notwendig:

<u>Ermittlung von Quelltextstrukturen, welche die Lösungen enthalten müssen:</u>

1. Identifikation der zur Realisierung notwendigen Hauptfunktionen
2. Identifikation der zugehörigen Subfunktionen
3. Identifikation notwendiger Elemente der (Sub-)funktionen und Erstellung des funktionalen Baums, welcher die Hauptfunktionen, Subfunktionen und die Elemente innerhalb dieser abbildet. Der funktionale Baum kann auch Implementierungsvarianten beinhalten.

<u>Definition der Suchmuster, welche aus diesen Code Strukturen resultieren:</u>

1. Zerlegung des funktionalen Baums in Teilbäume, welche durch einzelne Module spezifiziert werden sollen.
2. Abschätzung wie groß die Variantenvielfalt ist, mit welcher jedes dieser Teile/Module implementiert werden kann.
3. Aufgrund der Abschätzung in 2 und des festgelegten Klassifikationstyps muss entschieden werden, welches Abstraktionsniveau zur Suche nach dem Teilalgorithmus im einzelnen Modul erforderlich ist, bzw. welche Abstraktionstechniken verwendet und kombiniert werden sollen. Hieraus folgt die Auswahl der JPL Elemente, welche zur Musterdefinition eingesetzt werden.

Stufe 2: Konstruktion: In der Konstruktionsphase werden die zuvor ermittelten Suchmuster erstellt.

1. Erstellung der Suchmuster in einzelnen Modulen unter Einbezug der Implementierungsvarianten.
2. Festlegung, welche Module zu weiteren Modulen in Beziehung gesetzt werden können, um so komplexe Muster zu erzeugen. Diese Analyse bezieht sich auf die Implementierungsvarianten; so ist es möglich, dass für Modul A drei Suchmustervarianten erstellt werden und für Modul B nur zwei. Die Analyse kann nun ergeben, dass Modul A Variante 2 zwar mit Modul B Variante 1 kombinierbar ist, aber nicht mit Variante zwei. Somit ist nur die Erstellung des komplexen Musters aus MA V2 und MB V1 möglich, aber nicht MA V2 mit MB V2.
3. Im letzten Schritt werden die kombinierten Suchmuster erstellt, welche aus der Analyse im vorherigen Schritt resultieren.

Stufe 3: Ablauferstellung: Falls in den vorhergehenden Schritten mehrere Suchmuster erstellt wurden, so wird im Folgenden deren Anwendungsreihenfolge festgelegt. Einfluss auf die Reihenfolge können sowohl Abhängigkeiten zwischen Mustern, als auch Optimierungsaspekte haben.

- Mögliche Abhängigkeitsbeziehungen sind:
    - Muster A wird gefunden, daraus folgt Muster B wird ausgeführt.
    - Muster A wird gefunden, daraus folgt Muster B wird nicht ausgeführt.
    - Muster A wird gefunden, daraus folgt Muster B wird ausgeführt, alternativ wird Muster C ausgeführt.
- Zur Optimierung der Suche kann überprüft werden, ob Graphregeln, welche zur Realisierung der Muster erstellt werden, für weitere Muster wieder verwendet werden können. So könnte z.B. bei der Suche nach Muster A bereits Markierungen eingefügt werden, welche später bei der Suche nach Muster B hilfreich sind.

Der Vorteil der hier dargestellten Vorgehensweise liegt darin, dass als Basis für die Suchmustererstellung ausschließlich die in den Übungsaufgaben enthaltenen Anforderungen notwendig sind. Nachteilig wirkt sich aus, dass die Ableitung von Suchmustern aus Anforderungen einen sehr hohen zeitlichen Aufwand erfordern kann.

6.2. Bottom-Up über Syntaxanalyse

Im Gegensatz zur Top-Down Vorgehensweise bildet im Bottom-Up-Ansatz der Quellcode gegebener Aufgabenlösungen die Ausgangsbasis zur Suchmustererstellung. Mögliche Quellen dieses Codes sind:

- Vom Übungsleiter erstellte Beispiellösungen.
- Quellcode, der aus der Pilotierung der Übung resultiert (initiale Bearbeitung der Übung durch eine kleine Gruppe, z.B. Assistenten eines Universitätslehrstuhls).
- Quellcode der Lösungen von ähnlichen älteren Aufgaben.
- Quellcode, welcher typische Lösungen zu speziellen Teilaspekten der Übung beinhaltet.
- Quellcode aus Vorlesungen.

Da die gegebene Lösungsmenge in den meisten Fällen nicht alle Implementierungsvarianten umfassen wird, ist im Folgenden evtl. ein Abstraktionsschritt von den gegebenen Lösungen hin zu allgemeineren Suchmustern notwendig. Die Entscheidung, ob durch diesen Schritt die relevanten Lösungsvarianten erfasst werden oder ob weitere Muster erstellt werden müssen, obliegt dem Übungsleiter.

Hieraus folgt, dass die auf gegebenem Quelltext basierende Suchmustermenge sowohl sehr detaillierte Muster als auch generische Muster umfasst. Der Einsatz dieser generischen Muster beinhaltet die Möglichkeit des Auftretens von *false positives*. Somit ist je nach Umfang der Quelltextbasis zu bewerten, ob entweder generische Suchmuster erstellt und die Möglichkeit von *false positives* in Kauf genommen wird oder keine allgemeinen Muster erstellt werden, unter dem Risiko, dass *false negatives* für Lösungen erzeugt werden, welche nicht in der Quelltextbasismenge enthalten sind. Eine umfangreiche Menge an Lösungen würde die Entscheidung für die zweite Option nahe legen, da mit nur wenigen Lösungen zu rechnen ist, welche nicht bereits erfasst wurden. Bei einer nur geringen Anzahl von Beispiellösungen sollte eher die erste Vorgehensweise gewählt werden, um eine hohe Anzahl an *false negatives* zu vermeiden.

Vorgehensweise zur Erstellung von Mustern aus gegebenem Quellcode:

1. Identifikation typischer Module für einzelne Lösungsteile, z.B. Methoden in welchen jeweils eine Anforderung realisiert wird (z.B. Löschung eines Listenelements) und der entsprechenden JCG-Elemente, welche notwendig sind zur Spezifikation eines detaillierten Musters.
2. Für jedes Modul, welches abstrahiert werden soll (Entscheidungsprozess s. oben): Abstraktion von den detaillierten JCG-Elementen, z.B. Spezifikation einer Variablen, aber ohne Angabe des Typs, oder nur Spezifikation des Datenflusses für diese Variable, aber nicht die Variable selbst.
3. Kombination der Module zu komplexen Suchmustern

Die Entscheidung, ob diese Methode eingesetzt wird, ist abhängig vom Umfang der gegebenen Quelltextbasis und der gegebenen Zeit zur Suchmustererstellung. Falls zur Mustererstellung nur ein geringer Zeitraum zur Verfügung steht, kann auf die im Folgenden beschriebene Methode „Bottom-Up über Variablenabhängigkeitsbetrachtung“ zurückgegriffen werden.

6.3. Bottom-Up über Variablenabhängigkeitsbetrachtung

Die Vorgehensweise „Bottom-Up über Variablenabhängigkeitsbetrachtungen“ setzt Beispiellösungen zu der gegebenen Übungsaufgabe voraus. Im Gegensatz zur Vorgehensweise in Kapitel 6.2. steht hier nicht die syntaktische Struktur der Lösung, sondern die Betrachtung des Daten- und Kontrollabhängigkeitsgraphen im Vordergrund der Musterspezifikation. Diese Vorgehensweise geht davon aus, dass über die Betrachtung typischer Variablenabhängigkeiten in Übungslösungen mit geringerem Aufwand als in Kapitel 6.2. vorgestellt Suchmuster erstellt werden können, welche bereits eine Vielzahl möglicher Lösungen abdecken. Um Variablenabhängigkeiten innerhalb einer gegebenen Lösung zu analysieren und nachfolgend die für eine Anforderung benötigten AJSDG Elemente zu extrahieren, kann die Technik des Slicings verwendet werden, welche in Kapitel 5.11. beschrieben wurde.

Beispielaufgabe: Erstellen Sie eine Funktion, welche aus den Parametern a, b und der lokalen Variablen c, die Summe bildet.

Nutzen Sie das Codefragment:

```
public int (int a, int b){
    int c;
...}
```

Eine initiale Anforderung dieser Aufgabe ist die Abhängigkeit des Rückgabewerts der Methode von den Parametern a, b und c. Um diese Anforderung zu testen genügt es als Suchmuster den hierfür notwendigen Datenfluss zu erstellen.

Um auch Lösungen mit Hilfsvariablen im Suchmuster zu erfassen, wird im Folgenden ein Graph mit transitiven Datenabhängigkeitskanten verwendet:

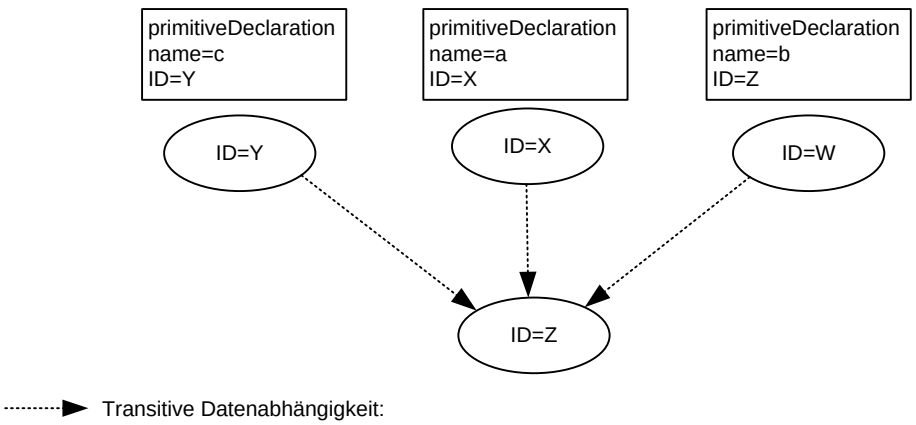


Abbildung 67: Spezifikation von Datenabhängigkeiten

Beispiellösung:

```
public void (int a, int b){
    int c;
    System.in(c);
    hilf1=a;
    int z = hilf1+b+c;
    return z;
}
```

Nachdem diese grundlegenden Datenabhängigkeiten modelliert wurden, kann das Suchmuster im nächsten Schritt durch weitere Elemente des JCG ergänzt werden, so dass eine größere Genauigkeit des Suchmusters bezogen auf die Gesamtanforderung entsteht. Die Abbildung unten detaillert den Ausdruck des Knotens mit der ID *Z* dahingehend, dass hier drei Variablen addiert und das Ergebnis einer Variablen zugewiesen wird.

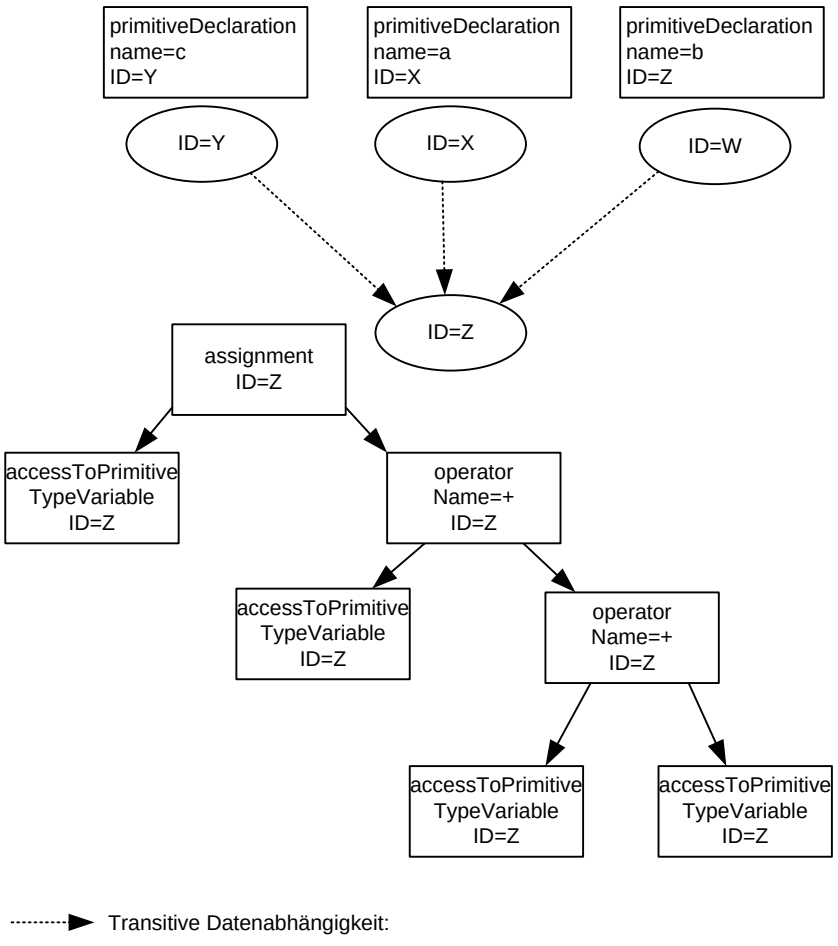


Abbildung 68: Spezifikation mit zusätzlicher JCG-Struktur

Kennzeichnend für diese Vorgehensweise ist, dass die Mustererstellung mit der Spezifikation von Daten- und/oder Kontrollflussabhängigkeiten beginnt, und nachfolgend durch das Hinzufügen von JCG Elementen konkretisiert wird.

6.4. Bottom-Up mittels eines gegebenen Musterkataloges

Die zugrunde liegende Idee dieses Vorgehenstyps ist die Verwendung eines Katalogs, welcher Suchmuster zu typischen Aufgabenlösungen bzw. den Lösungen von Teilaufgaben enthält. Ausgehend von diesen Mustern werden komplexe Suchmuster zur gegebenen Aufgabe erstellt. Hierzu können die zuvor beschriebenen Vorgehensweisen verwendet werden.

Prozess der Mustererstellung:

1. Die Ausgangsbasis für diese Vorgehensweise bildet ein Katalog, welcher Anforderungen und hierzu erstellte Lösungen enthält, denen entsprechende Suchmuster zugeordnet sind.
2. Initial wird geprüft, ob Anforderungen der aktuell gegebenen Übungsaufgabe strukturell den Anforderungen entsprechen, welche im Katalog enthalten sind. Dies bedeutet, die erwarteten Lösungen beinhalten ähnliche Strukturen.
3. Nachfolgend werden die diesen Anforderungen zugeordneten Suchmuster ermittelt.
4. Im Folgenden wird untersucht, in wieweit weitere Muster erstellt werden müssen (Vorgehensweisen hierfür s. Abschnitte 6.1, 6.2, 6.3), um die durch die Übungsaufgabe gegebenen Anforderungen abzudecken.
5. Nachdem alle einzelnen Muster erstellt wurden, wird untersucht ob die Suchmuster miteinander kombiniert werden können.
6. Zuletzt wird der Ablauf der Mustersuche festgelegt.

Kataloge sind im Rahmen der Suchmustererstellung ein geläufiges Hilfsmittel. So wird z.B. in [NSW+02] ein Patternkatalog verwendet, um über Graphmuster Design-Pattern zu ermitteln.

6.5. Zusammenfassung

In diesem Kapitel wurden vier Vorgehensweisen vorgestellt, welche zur Erstellung von Suchmustern eingesetzt werden können. Die Entscheidung, welche Vorgehensweise gewählt wird, hängt vom gegebenen Kontext ab. Stehen keine Beispiellösungen zur Verfügung, so kann auf die hier dargestellte funktionale Analyse zurückgegriffen werden. Sind bereits (ähnliche) Lösungen verfügbar, so kann eine Bottom-Up-Methode gewählt werden. Der Typ der Methode lässt sich aus der Anzahl der gegebenen Lösungen ableiten. Während im Rahmen einer kleinen gegebenen Lösungsmenge eher Bottom-Up mittels Variablenabhängigkeiten oder Syntaxanalyse angewendet wird, kann bei einer großen Basismenge auf einen Katalog zurückgegriffen werden.

Um den notwendigen Detaillierungsgrad eines Suchmusters zu bestimmen, sollten folgende Aspekte betrachtet werden:

1. Wie viel Zeit steht für die Musterspezifikation zur Verfügung?

   Aus einem höheren Abstraktionsgrad ergibt sich eine geringere Musterspezifikationszeit, da nicht so viele Elemente betrachtet werden müssen.

2. Wie viel Zeit steht zur Prüfung der Aufgaben zur Verfügung?

   Je genauer ein Suchmuster spezifiziert wird, desto unwahrscheinlicher ist das Auftreten von *false positives* und *false negatives*. Dies bedeutet eine stichprobenartige manuelle Prüfung der bereits automatisiert getesteten Aufgaben kann evtl. entfallen.

3. Wie groß ist der Umfang der Lösungsmöglichkeiten?

   Bei einer geringen Anzahl ist eine genauere Spezifikation auch in begrenzter Zeit möglich, während bei vielen möglichen Implementierungsvarianten eine abstraktere Suchmusterspezifikation notwendig ist.

4. Welche Konsequenzen ergeben sich aus dem Auftreten von *false positives* bzw. *false negatives*?

   Wird von einem Testverfahren ausgegangen, in dem automatisch als korrekt erkannte Muster nicht weiter untersucht werden, sollte eine möglichst detaillierte Spezifikation verwendet werden, während ein Ansatz, welcher weitere Nachprüfungen (z.B. manuelle Stichprobenverifikation oder dynamische Tests) mit einschließt, abstraktere Suchmuster erlaubt.

7. Automatisierte Testumgebung

In diesem Kapitel wird eine Applikationsumgebung beschrieben, welche die Automatisierung des in Kapitel 5. vorgestellten Prozesses von der Musterspezifikation bis zur Mustersuche in Java Quellcode unterstützt. Die Umgebung wird bezogen auf die Domäne der Prüfung von Übungsaufgaben im Rahmen der universitären Lehre dargestellt.

Das folgende Kapitel zeigt den allgemeinen Arbeitsablauf, welcher von der Aufgabenspezifikation bis zur automatisierten Bewertung der Lösungen der Studenten durchlaufen wird.

Kapitel 7.2. erläutert das Grobdesign eines Systems, welches die Spezifikation von JPL Mustern unterstützt, und diese Muster nachfolgend automatisiert in Graphregeln transformiert, so dass die Mustersuche über das im Folgekapitel aufgeführte Testsystem ausgeführt werden kann.

In Kapitel 7.3. wird die realisierte Umgebung zur Ausführung der Mustersuche über den einführend dargestellten Ablauf vorgestellt. Diese Umgebung wurde im Rahmen verschiedener Testate und Übungen am Lehrstuhl „Specification of Software Systems“ der Universität Duisburg-Essen eingesetzt und in den Veröffentlichungen [KG06] und [GSB08] beschrieben.

7.1. Arbeitsablauf für die Prüfung von Übungslösungen

Der Arbeitsablauf der Testumgebung zur Mustersuche besteht aus manuellen vorbereitenden Aktivitäten und automatisierten Funktionalitäten, welche durch Tools abgedeckt werden. Der Prozess beinhaltet drei Rollen:

1. Übungsleiter: Ein oder mehrere Übungsleiter erstellen die Übungen und die zugehörigen Spezifikationen der Suchmuster unter Zuhilfenahme der JPL (zu Vorgehensweisen s. Kapitel 6.). Nachfolgend werden die entsprechenden Muster aufgrund des in dieser Arbeit präsentierten Ansatzes abgeleitet und implementiert. Ein Grobdesign zur Automatisierung dieser Tätigkeit wird in Kapitel 7.2. vorgestellt.
2. Student: Der Student sendet den Quellcode seiner Aufgabenlösung an den Server, auf dem automatisch die Überprüfung des Codes angestoßen wird. Nach der Durchführung der automatisierten Tests erhält der Übungsleiter eine Rückmeldung, welche die entsprechenden Bewertungen der einzelnen Lösungen enthält. Dieser kann nun weitere manuelle Korrekturen vornehmen, um *false negatives* und/oder *false positives* auszuschließen.

   Nach der Durchführung der Tests erhält der Student eine Rückmeldung, welche die in seiner Lösung gefundenen Fehler enthält. Nach der Korrektur seiner Lösung kann der Student seine Lösung nochmals überprüfen lassen.
3. Server: Dem Server wird der Quellcode der Lösung übergeben. Es erfolgt die automatisierte Überprüfung der Aufgabenlösung über die in Kapitel 7.3. beschriebene Toolsequenz.

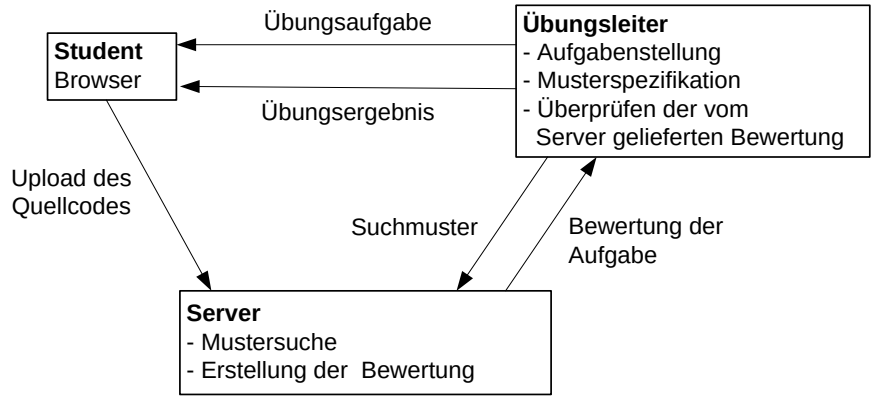


Abbildung 69: Arbeitsablauf zur automatisierten Prüfung von Übungsaufgaben

Um die Mustersuche und anschließende Bewertung der Aufgaben durchführen zu können, werden auf dem Server die Graphregeln, die Graphregelabfolge sowie die notwendigen Tools zur Graphverarbeitung, welche im Kapitel 7.3. beschrieben werden, installiert.

7.2. Systemumgebung zur Unterstützung der Spezifikation von Suchmustern

Dieses Kapitel beschreibt das Grobdesign eines Systems, welches die Spezifikation von JPL-Mustern unterstützt, und diese Muster nachfolgend automatisiert in Graphregeln transformiert über die die Mustersuche durchgeführt wird. Ausgehend von den Anforderungen wird das Grobdesign dargestellt, welches nachfolgend in die Testumgebung integriert wird.

Ein Nutzer dieses Tools (der Übungsleiter) muss zum Einsatz der JPL lediglich deren in Kapitel 4. beschriebenen konzeptionellen Aufbau und die zu verwendenden Syntaxelemente kennen. Vorgehenshinweise zum Aufbau des Musters können Kapitel 6. entnommen werden. Ein weitergehendes Verständnis der Graphtransformation ist für den Spezifikationsprozess und die Durchführung der Mustersuche nicht notwendig.

Anforderungen
In diesem Abschnitt werden die Anforderungen definiert, welche das System zur Unterstützung der Spezifikation von JPL-Mustern erfüllen muss.

Die Spezifikation und nachfolgende Ausführung eines Suchmusters lässt sich aus Sicht des Nutzers in vier grundlegende Schritte unterteilen:

1. Spezifikation der einzelnen Module eines Musters
2. Komposition der einzelnen Module zu einem Gesamtmuster
3. Ableitung des JPL-Graphen in die Mustervarianten
4. Durchführung der Mustersuche

Anforderungen zu 1:

Da sich ein Modul in 4 Sektionen aufteilt (Import, Körper, Export, Identifikator), wird zur Spezifikation der Module über eine GUI (Graphical User Interface) ein Grundgerüst dieser Bereiche vorgegeben, in welche die Elemente der JPL eingetragen werden können.

Anforderungen zu 2:

Die Komposition der einzelnen Module erfolgt über deren Schnittstellen. Hieraus folgt, dass die GUI des Tools zur Komposition den Fokus auf die Darstellung dieser Schnittstellen und die hier eingefügten Elemente legen muss. Der Nutzer spezifiziert in dieser Umgebung die Import-/Export-Beziehungen durch die Erzeugung von Verbindungskanten zwischen den Knoten in den Schnittstellensektionen. Das Tool muss nach erfolgter Komposition eine Konsistenzprüfung durchführen.

Anforderungen zu 3:

Nachdem der abstrakte JPL-Graph erstellt wurde, wird dieser automatisiert in verschiedene Muster-Varianten abgeleitet. Das Ergebnis der Ableitung ist eine Datei welche die Regeln enthält, die zur Mustersuche über die in Kapitel 7. vorgestellte Testumgebung ausgeführt werden können. Hieraus folgt, dass die Graphregeln ausschließlich Elemente aus den Typgraphen des JCG, des AJSDG und Markierungsknoten verwenden dürfen.

Anforderungen zu 4:

Die Mustersuche erfolgt über die in Kapitel 7.3. vorgestellte Testumgebung.

Grobdesign

Da durch das Graphtransformationstool AGG eine API (Application Programming Interface) zur Verfügung gestellt wird, welche sowohl Zugriff auf die Graphtransformations-Engine, als auch auf die einzelnen grafischen Objekte der GUI des Tools bereitstellt, bietet es sich an diese API zu nutzen und AGG bezüglich der Anforderungen, welche für die Modulerstellung und Modulkonzeption spezifiziert wurden, zu erweitern.

Hierzu ist eine Komponente notwendig, welche den Frame erweitert, der den Arbeitsgraph darstellt. Dieser Frame muss in die oben beschriebenen einzelnen Sektionen unterteilt werden. Sobald ein Knoten in die jeweilige Sektion eingefügt wird, erhält das Attribut *Sektion* des Knotens den entsprechenden Wert. Weiterhin wird das Attribut *ModulNr* automatisch gesetzt und dem Knoten bzw. der Kante eine ID zugeordnet.

Die Komposition mehrerer Module erfolgt ebenfalls über die GUI des Tools. Es werden lediglich die Sektionen Import, Export und Identifikator sowie deren Inhalte angezeigt, so dass die Elemente in diesen Schnittstellensektionen miteinander verbunden werden können. Falls am Ende des Spezifikationsprozesses nicht alle Attribute vom Nutzer mit Werten belegt wurden, so werden den unbelegten Attributen Default-Werte zugeordnet. Vom Nutzer eingegebene Attributwerte werden im Weiteren in die Suchmuster übernommen, während die Default-Werte lediglich dazu verwendet werden einen vollständigen JPL-Graphen zu erzeugen, über den regelbasierte Graphtransformationen

ausgeführt werden können. Der JPL-Graph wurde somit vollständig spezifiziert und wird nachfolgend in einer Datei abgelegt.

Im nächsten Schritt werden die Graphregeln und deren Ablaufspezifikation für die Testumgebung erstellt. Hierzu werden auf den zuvor gespeicherten JPL-Graphen die Regeln und Regelabläufe angewendet, welche in den Kapiteln 5.4., 5.5., 5.6. und 5.7. beschrieben wurden, so dass automatisiert die Suchmustervarianten, wie sie durch die JPL definiert sind, erstellt werden.

Die Auswahl der zu verwendenden Regeln kann vom Nutzer gesteuert werden, indem er folgende Informationen angibt:

1. Existieren JCG- oder AJSDG Knoten in einer Importsektion, die nicht mit weiteren Knoten verbunden sind?
2. Enthält der Graph direkte JCG- oder AJSDG-Übergaben?
3. Enthält das Suchmuster Verbindungen zwischen den Körper- und Schnittstellensektionen?
4. Enthält das Muster JPL-Knoten und/oder JPL-Schnittstellenstrukturen?
5. Welche Übergabevarianten sollen zur Ableitung der JPL-Schnittstellenstrukturen angewendet werden?
6. Welche Suchvariante (sektionsbasiert oder vollständig) soll verwendet werden?

Alternativ können die Antworten auf die Fragen eins bis vier durch eine automatische Analyse des JPL-Graphen ermittelt werden und/oder für Punkt fünf alle Regelsätze durchlaufen werden, die in den Kapiteln 5.4, 5.5. und 5.6. beschrieben wurden, so dass eine Vielzahl an Mustervarianten erstellt wird.

Nachdem für alle geforderten Übergabevarianten die Zielgraphen erzeugt wurden, wird im nächsten Schritt ein Regelsatz erstellt, welcher die Suche nach den Mustervarianten realisiert. Der Prozess, nach dem dieser Regelsatz abläuft, ist in den Kapiteln 5.9. und 5.10. beschrieben.

Zuvor muss der Nutzer entscheiden, ob die Musterableitung sektionsbasiert oder vollständig erfolgen soll. Entsprechend dieser Entscheidung werden unterschiedliche Regelsätze erstellt. Die technische Übertragung der Implementierungsvarianten, d.h. der erzeugten Graphen in die Ablaufdatei zur nachfolgenden Musteridentifikation wird entsprechend der gewählten Musterableitung ausgeführt.

Überlegungen zum User Interface:

Das modulbasierte Interface der JPL setzt, wie auch die Algorithmen zur Musterableitung und Mustersuche, auf einer visuellen Darstellung über Graphen auf. Hierbei steht die Überlegung im Vordergrund, dass die Suchmuster zu Übungsaufgaben stark durch die Spezifikation der Beziehung der Elemente untereinander, und nicht so sehr durch die Suche nach einzelnen isolierten Elementen oder Elementgruppen geprägt sind. Diese Beziehungen, und besonders die Kombination aus SDG- und JCG-Elementen, lassen sich über Graphen gut darstellen.
Falls sich zukünftig der Fokus auf Anwendungen, die sich stärker auf die Suche nach einzelnen Elementen konzentrieren, verlagern sollte, oder Übungsaufgaben mit sehr

starren Vorgaben, z.B. die Reihenfolge der einzelnen Anweisungen betreffend, überprüft werden müssen, so sind auch andere Interfaces, wie z.B. ein Ansatz über Lückentexte, möglich. Die so spezifizierten Suchmuster können nachfolgend auf die in dieser Arbeit präsentierte Graphstruktur übertragen werden, um sowohl den Vorgang der Musterableitung als auch der Mustersuche aus dieser Arbeit zu nutzen.

Technisches Design des Ableitungsprozesses vom JPL-Graphen zur Datei, welche die Regeln zur Ausführung der Mustersuche enthält.

Im Folgenden wird ein technisches Design zur Erstellung der Datei, welche die Regeln für den Ablauf der Mustersuche enthält (Zieldatei), vorgestellt. Initial werden für den JPL-Graphen verschiedene Dateien erzeugt, welche die Implementierungsvarianten für das zu suchende Muster enthalten. Hierzu wird für jede Variante zu einer JPL-Schnittstellenstruktur ein vorgegebener Regelsatz auf dem JPL-Graphen ausgeführt.

Nachfolgend wird die Zieldatei erstellt, indem

1) verschiedene Regeln aus Templates in die Zieldatei kopiert werden und z.T. anhand der Varianten parametrisiert werden.
2) die Suchmuster der Varianten in die Zieldatei eingetragen werden, wobei zwischen vollständiger Mustersuche und sektionsbasierter Mustersuche unterschieden werden muss.

Der im Folgenden dargestellte Verlauf zeigt die Erstellung der Datei zur Ausführung der Mustersuche. Es werden drei Varianten der Mustersuche unterschieden:

1. Die Suche einer vollständigen Mustervariante in einem Schritt.
2. Die Suche einer Variante durch eine kombinierte Suche der sektionsbasierten Teilmuster. Die Übergabestrukturen werden aus dem Gesamtmuster hergeleitet. Verbindungen zwischen Knoten der Körpersektion und JPL-Schnittstellenknoten werden in die Mustererstellung mit einbezogen.
3. Die Suche einer Variante durch eine kombinierte Suche der sektionsbasierten Teilmuster. Die Übergabestrukturen werden aus einem Katalog entnommen. Verbindungen zwischen Knoten der Körpersektion und JPL-Schnittstellenknoten werden in die Mustererstellung nicht mit einbezogen. JPL-Schnittstellenstrukturen werden separat betrachtet und durch JCG-Strukturen ersetzt.

Die Varianten werden in Kapitel 5.3. erläutert und miteinander verglichen. Die folgende Abbildung stellt den Prozess der Erstellung der Zieldatei für die drei Varianten der Mustersuche graphisch dar. Die in der Abbildung aufgeführten Einzelschritte werden im Rahmen einer detaillierten Prozessbeschreibung in den nächsten Abschnitten erläutert.

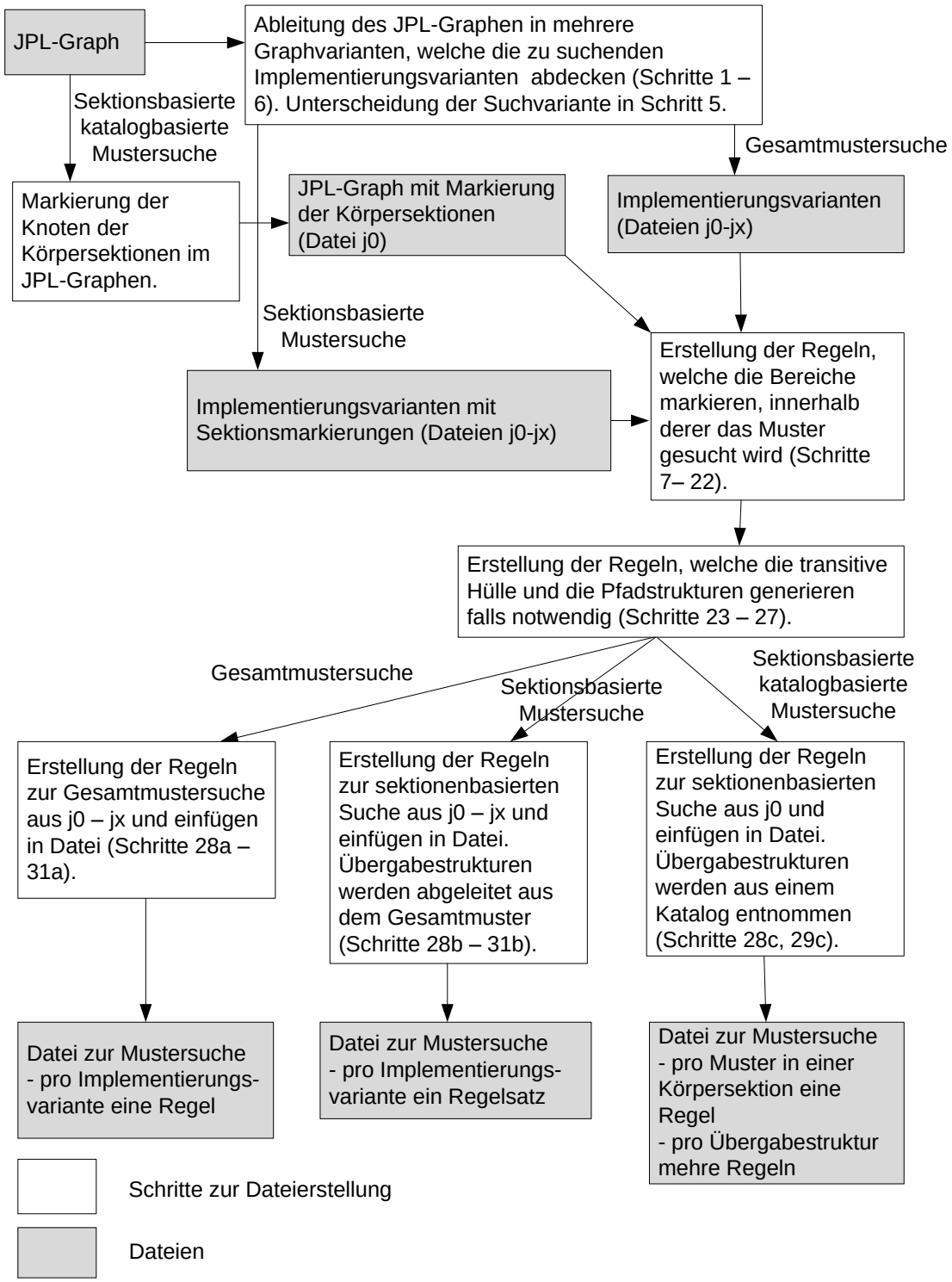


Abbildung 70: Prozess zur Erstellung der Datei, über welche die Mustersuche durchgeführt wird

Die Abbildung zeigt für die zuvor dargestellten drei Varianten der Mustersuche den Prozess der Musterableitung. Für jede Variante wird jeweils eine oder mehrere Dateien generiert (j0 oder j0-jx), aus welchen die entsprechen Regelsätze zur Mustersuche erstellt werden. Zu beachten ist, dass zur Generierung der Regeln für die Markierung der Suchräume und die Erstellung der Transitiven Hülle für jede Variante die gleichen

Schritte ausgeführt werden. Die Erstellung der Regelsätze zur Mustersuche unterscheidet sich hiernach je nach Variante.

Für das technische Design wird davon ausgegangen, dass die verschiedenen Graphen und Graphregeln in XML (Extensible Markup Language)-Dateien abgelegt werden. Ein großer Teil des Ablaufs der Generierung der Zieldatei wird über die Schritte des Parsens einer Datei, der nachfolgenden Suche nach einem Elementtyp welcher ein vorgegebenes Attribut enthält, und dem Kopieren dieses Elements und evtl. einiger mit diesem verbundenen Elemente in eine Struktur der Zieldatei durchgeführt.

Das XML-Format (Schema) muss folgende Eigenschaften erfüllen:

- Es existiert ein Datentyp für Knoten
- Es existiert ein Datentyp für Kanten
- Es existiert ein Datentyp für eine Regel welcher die linke und rechte Regelseite umfasst. Die linke und rechte Regelseite können explizit identifiziert werden (als eigener Datentyp oder Subtyp der Regel)
- Es existiert ein Datentyp für den Graphen
- Alle Elemente der verschiedenen Datentypen in der XML-Datei sind über eine ID eindeutig identifizierbar.

Das von AGG verwendete ggx-Format kann als Beispiel für diesen Dokumenttyp herangezogen werden.

Kopieren und IDs

In den folgenden Prozessschritten werden Elemente der Datentypen *Knoten, Kante, Regel* und *Graph* von einer Quelldatei in eine Zieldatei kopiert. Um Konflikte mit den bestehenden IDs in der Zieldatei zu vermeiden, wird folgendes Vorgehen vorgeschlagen:
Wenn ein Element in eine Datei kopiert wird, so wird diesem aufsteigend eine neue ID zugewiesen, d.h. es wird die höchste ID der Elemente eines Datentyps in der Zieldatei ermittelt, um 1 erhöht und diese dem neu hinzuzufügenden Element zugewiesen.
Falls kein einzelnes Element sondern eine Struktur kopiert wird (z.B. zwei durch eine Kante verbundene Knoten), so werden die IDs entsprechend konsistent sowohl im Element welches durch die ID identifiziert wird (z.B. ein Knoten), als auch in Elementen welche über diese ID auf das Element verweisen (z.B. eine Kante, welche über ein Attribut auf den Quell- oder Zielknoten verweist), geändert.

Template:
Ein Template enthält einen oder mehrere semantisch zusammenhängende Regelsätze.

Folgende Templates werden eingesetzt:

a. Template mit Regeln welche den JPL-Graphen um Hilfsstrukturen ergänzen, u.a. zur Markierung der Knoten der einzelnen Sektionen (Kapitel 11.2., Abbildung 97 bis Abbildung 101), zur Erstellung von Hilfsstrukturen zur Erkennung zusammenhängender AJSDG und JCG Knoten (Kapitel 11.2., Abbildung 125) und zur Sicherstellung des Importverhaltens (Kapitel 11.2., Abbildung 102 bis Abbildung 104).
b. Templates (b1 bis bx) mit Regeln zur Ableitung für je eine Implementierungsvariante (Hauptregeln für die verschiedenen Varianten s.

Kapitel 5.4., 5.5., 5.6., Hilfsregeln im Anhang Kapitel 11.2.). Templates mit Regeln zur Ableitung von direkten Übergaben (Kapitel 11.2., Abbildung 110 bis Abbildung 113).

c. Template mit Regeln zur Erstellung der Transitiven Hülle (Regeln s. Kapitel 5.8.)
d. Template mit Regeln zur Erstellung eines Pfades (Regeln s. Kapitel 5.9. und Kapitel 11.2., Abbildung 95 und Abbildung 96)
e. Template mit Regeln zur Markierung der Sektionen (Kapitel 11.2., Abbildung 87 bis Abbildung 94), Regeln zur Markierung der Bereiche außerhalb der Sektionen (Kapitel 11.2., Abbildung 102 bis Abbildung 104) und Regeln zur Löschung von Markierungsknoten und Schnittstellenmarkierungsknoten..
f. Template mit technischen Hilfsregeln zur Erweiterung des AJSDG und des JCG als Vorbereitung zur Mustersuche (Kapitel 11.2., Abbildung 123 bis Abbildung 127).

Der unten dargestellte Ablauf mit Schwerpunkt auf dem Parsen von XML Dateien, der Suche nach Knoten und Kanten mit bestimmten Attributwerten und dem Kopieren von Strukturen kann z.B. über ein Java-Programm realisiert werden. Die ersten sechs Schritte sind nur auszuführen, falls das JPL-Muster über Graphtransformationsregeln in verschiedene Mustervarianten abgeleitet werden soll. Die Schritte sind somit notwendig für die Suche nach dem vollständigen Muster und die sektionsbasierte Suche, welche auch die Beziehung zwischen Elementen der Körpersektion und der Schnittstellensektion betrachtet. Sie sind nicht notwendig, wenn mit einem Katalog von Übergabemustern gearbeitet wird. Hier werden nur die Knoten der Körpersektion der jeweiligen Bereiche markiert (Kapitel 5.7 Schritte 1, 2,4), wobei in Schritt 4 die Datei j0 erstellt wird.

Der abzuleitende JPL-Graph ist in der XML Datei h gespeichert.

Der folgende Prozessablauf wird in Blöcke unterteilt dargestellt, die semantisch zusammenhängende Anweisungen enthalten. Die Anweisungen werden entsprechend ihrer Nummerierung sequentiell durchlaufen.

Ableitung des abstrakten JPL-Graphen in Graphvarianten, welche die zu suchenden Implementierungsvarianten abdecken.
1) Kopiere den JPL-Graphen aus Datei h in den Regelsatz a, führe die Regeln aus und speichere den sich ergebenden Graphen in Datei i. 2) Wähle ein Template welches die Ableitung einer JPL-Schnittstellenstruktur in die gewünschte flache Struktur realisiert (b1 bis bx), kopiere den JPL-Graphen (i) in diesen Regelsatz und führe die Regeln auf dem JPL-Graphen aus. Die Wahl des Templates kann manuell durch den Nutzer erfolgen oder automatisch durch einen Algorithmus ausgeführt werden, wenn z.B. Muster für alle möglichen Übergabevarianten erstellt werden sollen (sequentielle Abarbeitung aller Templates aus b). 3) Die zu suchende flache Mustervariante welche für diese JPL-Schnittstellenstruktur nur JCG-Elemente enthält, wurde erstellt und wird in einer XML-Datei (ix) gespeichert.

4) Falls der JPL-Graph weitere JPL-Strukturen enthält gehe zu 2, sonst gehe zu 5. 5a) Falls die sektionsbasierte Mustersuche gewählt wurde, markiere die einzelnen Sektionen (Kapitel 5.7. Schritte 1-4). In Schritt 4 wird eine Datei (j0, j1, j2, …jx) erstellt. 5b) Fall die Suchvariante über die vollständigen Muster gewählt wurde, erstelle für die abgeleitete Mustervariante eine eigene Datei (j0, j1, j2, …jx). 6) Falls eine weitere Mustervariante erstellt werden soll gehe zu 1, sonst gehe zu 7.

Im Folgenden wird der Prozess zur Erstellung der Datei, die zur Durchführung der in den Kapiteln 5.9 und 5.10 dargestellten Abläufe notwendig ist, beschrieben. Die leere Datei in welche die Regeln zum Ablauf der Mustersuche eingefügt werden, wird als g repräsentiert.

Die Regeln werden in der Reihenfolge ihres Aufrufs in der Datei g gespeichert. In AGG kann dies durch eine aufsteigende Nummerierung des Layers der Regeln sichergestellt werden.

Erstellung der technischen Hilfsregeln.
7) Kopiere die Regeln aus Template f nach g.

Erstellung der Regeln zur Bereichsmarkierung für Klassen und Interfaces.
8) Suche in der Datei j0 nach einem Knoten vom Typ *class, interface, abstractClass* und mit der Attributbelegung *sektion=Identifikator* und speichere den Inhalt seines Attributs *ModulNr*. Falls kein Knoten gefunden wird gehe zu 12, sonst setze das Attribut *sektion* auf *Identifikatorfound* und gehe zu 9. 9) Kopiere aus Template e die Regel zur Klassenmarkierung (Abbildung 89: Bereichsmarkierung für Klassen 1) in die Datei g. Parametrisiere den freien Knoten mit den Attributen des in 8 gefundenen Knotens. 10) Suche in j0 alle Knoten und Kanten mit der gespeicherten Modulnummer und der Attributbelegung *sektion=Identifikator* und kopiere diese nach g in die linke und rechte Regelseite der zuvor betrachteten Regel (vollständiges Mapping). Hierbei ist zu beachten, dass Kanten, welche eine Verbindung zu dem in 8 gefundenen Knoten besitzen, die richtige Quellknoten-ID oder Zielknoten-ID (erstellt in Schritt 9) zugewiesen wird. 11) Gehe zu 8.

Erstellung der Regeln zur Bereichsmarkierung für Methoden, Schleifen und Bedingungen.
12) Suche in der Datei j0 nach einem Knoten vom Typ *method, while, do, if, else, try, switch,* und mit der Attributsbelegung *sektion=Identifikator* und speichere den Inhalt seines Attributs *ModulNr*. Prüfe ob der Knoten der Zielknoten einer Kante vom Typ *Hierachiekante* ist. Falls ja setze das Attribut *sektion* auf *IdentifikatorHierarchisch* und wiederhole die Suche. Falls kein Knoten gefunden wird gehe zu 15, sonst gehe zu 13.
13) Kopiere aus Template e die Regel zur Markierung von „Blöcken die keine Klassen sind“ (Abbildung 87) in die Datei g. Parametrisiere den freien Knoten mit den Attributen des in 12 gefundenen Knotens.
14) Suche in j0 alle Knoten und Kanten mit der gespeicherten Modulnummer und der Attributbelegung *sektion=Identifikator* und kopiere diese nach g in die linke und rechte Regelseite der zuvor betrachteten Regel (vollständiges Mapping). Hierbei ist zu beachten, dass Kanten, welche eine Verbindung zu dem in 12 gefundenen Knoten besitzen, die richtige Quellknoten-ID oder Zielknoten-ID (erstellt in Schritt 13) zugewiesen wird. Gehe zu 12.
15) Falls in 8 oder 12 mindestens ein Knoten gefunden wurde kopiere die Regel zur Markierung der zu durchsuchenden Bereiche nach g (Abbildung 88).

Erstellung der Regeln zur Bereichsmarkierung für hierarchisch geschachtelte Bereiche.
16) Suche in der Datei j0 nach einem Knoten vom Typ *class, method, while, do, if, else, try, switch,* der Zielknoten einer Kante vom Typ *Hierarchiekante* ist und speichere den Inhalt des Attributs *ModulNr* des gefundenen Knotens. Falls kein Knoten gefunden wird, gehe zu 20.
17) Kopiere aus Template e die Regeln zur Markierung von hierarchisch abhängigen Bereichen (Abbildung 91) in die Datei g. Parametrisiere den freien Knoten welcher den abhängigen Bereich markiert, mit den Attributen des in 16 gefundenen Knotens. Parametrisiere den freien Knoten, welcher den übergreifenden Bereich markiert mit den Attributen des Knotens in j, welcher der Quellknoten der Hierarchiekante ist welche auf den gefundenen Knoten zeigt. Speichere das Attribut *Modulnr* des Quellknotens. Setze das Attribut *sektion* des Zielknotens auf *Identifikatorfound*.
18) Suche in j0 alle Knoten und Kanten mit der gespeicherten Modulnummer des Zielknotens und der Attributbelegung *sektion=Identifikator* und kopiere diese nach g in die linke und rechte Regelseite der zuvor betrachteten Regel (vollständiges Mapping). Hierbei ist zu beachten, dass Kanten, welche eine Verbindung zu dem Zielknoten besitzen, die richtige Quellknoten-ID oder Zielknoten-ID (erstellt in Schritt 17) zugewiesen wird.
19) Suche in j0 alle Knoten und Kanten mit der gespeicherten Modulnummer des Quellknotens und der Attributbelegung *sektion=Identifikator* und kopiere diese nach g in die linke und rechte Regelseite der zuvor betrachteten Regel (vollständiges Mapping). Hierbei ist zu beachten, dass Kanten, welche eine Verbindung zu dem Zielknoten besitzen, die richtige Quellknoten-ID oder Zielknoten-ID (erstellt in Schritt 17) zugewiesen wird. Gehe zu 16.
20) Falls in 16 mindestens ein Knoten gefunden wurde kopiere die Regel zur Markierung der zu durchsuchenden Bereiche nach g (Abbildung 88).

Erstellung der Regeln zur Markierung von Elementen außerhalb der Gültigkeitsbereiche.
21) Suche in der Datei j0 nach einer Kante welche einen Knoten vom Typ *MarkAll* als Quellknoten hat. Falls keine Kante gefunden wird gehe zu 23, sonst gehe zu 22. 22) Kopiere aus Template e die Regel welche die Elemente markiert die nicht in der zu untersuchenden Sektion liegen (Abbildung 102 oder Abbildung 103) nach g.

Erstellung der Regeln zur Generierung der Transitiven Hülle und des Pfades.
23) Suche in der Datei j0 nach einer Kante vom Typ *Pfad* und falls diese nicht gefunden wird, nach einer Kante vom Typ *transitive Hülle*. Falls eine Kante gefunden wird, gehe zu 24, sonst gehe zu 28. 24) Kopiere die Regeln zur Erstellung der Transitiven Hülle von c nach f. 25) Falls in 23 eine Pfadkante gefunden wurde gehe zu 26, sonst gehe zu 28. 26) Kopiere die Regeln zur Erstellung eines Pfades von d nach f. 27) Suche in der Datei j0 nach Knoten, welche Quelle und Ziel der Pfadkante sind und kopiere die Attribute des Quellknotens und des Zielknotens in die zu parametrisierenden Knoten der Pfadregeln (Abbildung 95 und Abbildung 96).

Während der Ablauf der Anweisungen für die obigen Schritte sequentiell erfolgte, sind die folgenden drei Anweisungsblöcke als Alternativ zu betrachten (s. Abbildung 70: DreiVarianten zur Mustersuche).

Erstellung der Regeln zur Identifikation des Musters im Quellcode, Variante vollständige Mustersuche.
28a) Füge den in jx (bei erstmaligem Aufruf ist x=0) gespeicherten Graphen als neue Regel in der linken und rechten Regelseite (vollständiges Mapping) ein. Falls es keine Kante zwischen einem Knoten der Sektion *Identifikator* und einem Knoten in einer weiteren Sektion gibt, werden alle Knoten und Kanten kopiert deren *sektion*- Attribut nicht mit dem Wert *Identifikator* belegt ist. Ansonsten werden auch die Knoten und Kanten der Sektion *Identifikator* kopiert, welche direkt oder indirekt mit einem Knoten einer anderen Sektion verbunden sind. Falls ausgeschlossen werden soll das Teilmuster der Körpersektionen in überlappenden Bereichen identifiziert werden, muss die Regel um NACs für alle Bereichsmarkierungsknoten ergänzt werden, die verhindern, dass jeweils zwei Bereichsmarkierungsknoten auf identische Knoten referenzieren. 29a) Füge den Ergebnisknoten auf der rechten Regelseite ein. 30a) Lösche die Datei jx. Falls eine Datei jx+1 existiert, öffne diese und gehe zu 28 31a) Speichere g.

Erstellung der Regeln zur Identifikation des Musters im Quellcode, Variante sektionsbasierte Mustersuche.
28b) Führe die in Kapitel 5.7, Schritte 5 - 7 beschriebene Regelfolge auf jx (bei erstmaligem Aufruf ist x=0) als Quelldatei, welche das Suchmuster enthält, und g als Zieldatei aus. 29b) Falls eine Datei jx+1 existiert, kopiere die Regeln zur Löschung der Markierungsknoten und Schnittstellenmarkierungsknoten (e) nach g, damit die Regeln des nachfolgenden Regelsatzes auf einen unmarkierten Graphen aufsetzen können. 30b) Lösche die Datei jx. Falls eine Datei jx+1 existiert, öffne diese und gehe zu 28 31b) Speichere g.

Erstellung der Regeln zur Identifikation des Musters im Quellcode, Variante sektionsbasierte, katalogbasierte Mustersuche.
Template k: Template mit Regeln zu den möglichen Implementierungsvarianten der JPL-Schnittstellenstrukturen (Katalog der Übergabevarianten). Die Regeln ergeben sich aus den jeweils rechten Regelseiten die für die Varianten in den Kapiteln 5.4., 5.5., 5.6. dargestellt wurden (Bsp. Kapitel 5.5. Abbildung 35, Kapitel 5.6. Abbildung 40 bis Abbildung 42, Abbildung 44 bis Abbildung 46). 28c) Führe den in Kapitel 5.7 Schritte 5 – 7 beschriebenen Prozess auf j0 als Quelldatei, welche das Suchmuster enthält, k als Katalog für die Übergabevarianten und g als Zieldatei aus. Dies bedeutet in Schritt 6 wird die Alternative des Katalogs gewählt. 29c) Speichere g.

Überlegungen zur Performance des Ableitungsprozesses:

- Der Prozess wird pro JPL-Graph nur einmal durchlaufen, wodurch er für den Gesamtprozess der Mustererstellung und Mustersuche nicht so performancekritisch einzustufen ist wie der Ablauf der Mustersuche der für jede zu untersuchende Übungslösung durchzuführen ist. Der Fokus des Prozesses liegt auf der Stabilität der zu erstellenden Applikation. Hierbei wird besonders die Auslastung des Hauptspeichers betrachtet, welche sowohl im Rahmen des Vorhaltens großer Graphen als auch durch deren Transformation stark ansteigen kann.
- Zur Ableitung des JPL-Graphen werden Regelsätze aus verschiedenen Templates genutzt, die semantisch zusammenhängende Regeln enthalten. Diese Unterteilung ist erfolgt, damit immer nur die für den jeweiligen Schritt notwendigen Regeln in den Hauptspeicher geladen werden müssen.
- Die Muster zu den verschiedenen Implementierungsvarianten werden jeweils in eine eigene Datei gespeichert, um zu verhindern, dass die Varianten im Hauptspeicher gehalten werden müssen und dieser zu stark beansprucht wird. Die Ladezeit und Zeiten zur Speicherung der Datei verringern die Performance

zugunsten einer geringeren Belastung des Hauptspeichers. Weiterhin sind so bei einem Absturz des Systems die erstellten Regelvarianten nicht verloren, sondern es kann an dem Punkt fortgefahren werden, an welchem der Fehler auftrat.

- Die Entscheidung am Ende des Prozesses alle für die Mustersuche notwendigen Regeln in einer Datei abzulegen, über welche die Mustersuche durchgeführt wird, erfolgte aus der Überlegung, dass sich bei der sequentiellen Abarbeitung mehrerer Dateien durch AGG viele Regeln wiederholt hätten (technische Regeln, Regeln zu Bereichsmarkierungen, Erstellung der transitiven Hülle) und die Prüfung der Aufgaben performancekritisch ist, also möglichst viele Daten schon im Hauptspeicher zugreifbar sein sollten.

Beispiele zur Verdeutlichung des Ableitungsprozesses

Im Folgenden wird für zwei Beispiele der Ablauf der Ableitung einer Suchmustervariante aus dem JPL-Graphen, und die nachfolgende Erstellung der Datei, welche zur Mustersuche genutzt wird, über Screenshots dargestellt. Die Beispiele sind aus den Implementierungen entnommen, die zur Realisierung der Anwendungsbeispiele in Kapitel 8 genutzt werden.

Die Screenshots zeigen den JPL-Graphen, eine für das jeweilige Beispiel automatisiert abgeleitete Suchmustervariante und einige aus der Variante resultierende Regeln, welche manuell in die Datei eingefügt wurden, über welche die Mustersuche durchgeführt wird. Die Screenshots zum JPL-Graphen und der Suchmustervariante stammen aus den Dateien *Ableitung des JPL-Graphen* und *Ablauf der Mustersuche* welche zur Durchführung des jeweiligen Anwendungsbeispiels erstellt wurde. Die Dateien sind der Arbeit auf einer DVD beigelegt. Der Ablauf der Musterableitung und Mustersuch wurden weiterhin in Videos festgehalten (s. Kapitel 11.3. zur Auflistung der Anwendungsfälle, deren Ablauf aufgezeichnet wurde. Die Videos befinden sich auf der beiliegenden DVD.). Die automatisierte Mustersuche selbst ist nicht Inhalt dieses Kapitels und somit im Ablauf nicht erfasst.

Beispiel zur Suche über ein Gesamtmuster

Die Anwendung des oben beschriebenen Ableitungsprozesses für die Variante der vollständigen Mustersuche wird im Folgenden für das Anwendungsbeispiel *Aufgabe1 Muster2 Parameterübergabe zwei Bedingungen* dargestellt. Dieses Beispiel wird in Kapitel 8.1. als zweites Muster beschrieben und der Aufgabenkontext erläutert.

Der folgende Screenshot zeigt den JPL-Graphen, welcher in der Graphsektion von AGG beschrieben wurde. Ausgehend von diesem abstrakten JPL-Graphen wird über Ableitungsregeln eine Suchmustervariante generiert, welche nach einer konkreten Implementierungsvariante des Musters sucht. Die Regeln wurden für dieses Beispiel manuell erstellt.

Hinweis: Das Attribut *ID* dient nur zur eindeutigen Identifizierung der im JPL-Graphen spezifizierten Elemente, um die Zuordnung von *AJSDG*- zu *JCG*-Knoten zu beschreiben. Die für den Ableitungsprozess (Kopieren von Teilgraphen in Regeln) notwendige Identifizierung der Elemente erfolgt über die von AGG automatisch generierten IDs, welche in der GGX-Datei hinterlegt sind.

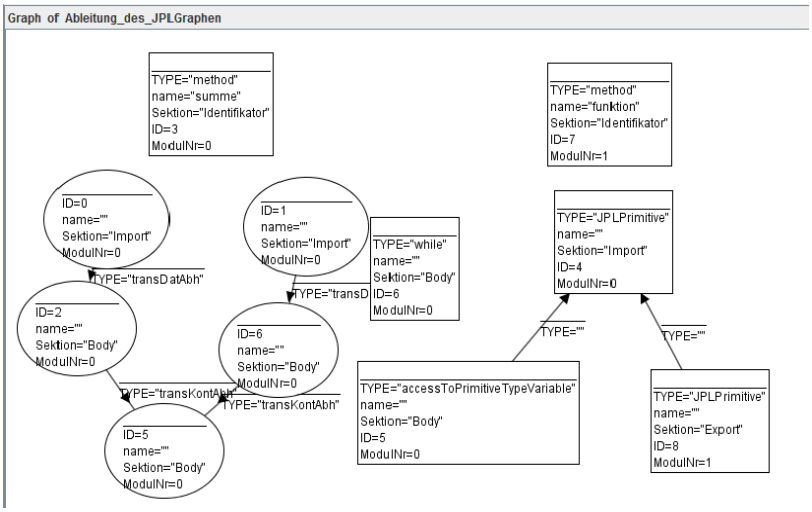


Abbildung 71: JPL-Graph des Suchmusters

Der folgende Screenshot zeigt die Suchmustervariante, welche nach dem automatisierten Ablauf der Ableitungsregeln entsteht. Dieser Vorgang wird im Block *Ableitung des abstrakten JPL-Graphen in Graphvarianten, welche die zu suchenden Implementierungsvarianten abdecken.* beschrieben. In Abbildung 70 ist dies im Kasten *Ableitung des JPL-Graphen in mehrere Graphvarianten, welche die zu suchenden Implementierungsvarianten abdecken (Schritte 1 – 6). Unterscheidung der Suchvariante in Schritt 5.*dargestellt.

In dieser Variante wird in der Methode *summe* ein Wert verwendet, der von der Methode *funktion* über eine *return*-Anweisung übergeben wird (Ersetzung der JPL-Schnittstellenstruktur mit der konkreten Variablenübergabe, welche eine *return*-Anweisung nutzt). Weiterhin wurden die Knoten der beiden Module (zu erkennen am Attribut *ModulNr)* mit Bereichsmarkierungsknoten markiert und die Knoten, welche importiert werden, über eine Beziehung zum Knoten vom Typ *MarkAll* gekennzeichnet.

Über diese Variante wird z.B. folgendes Quelltextsegment identifiziert:

```
public double summe(int von, int bis){
i=von;
if  (i>0){
while (i <=bis){
                    … funktion(i);
}}

public double funktion …{
double u;
return u;
}
```

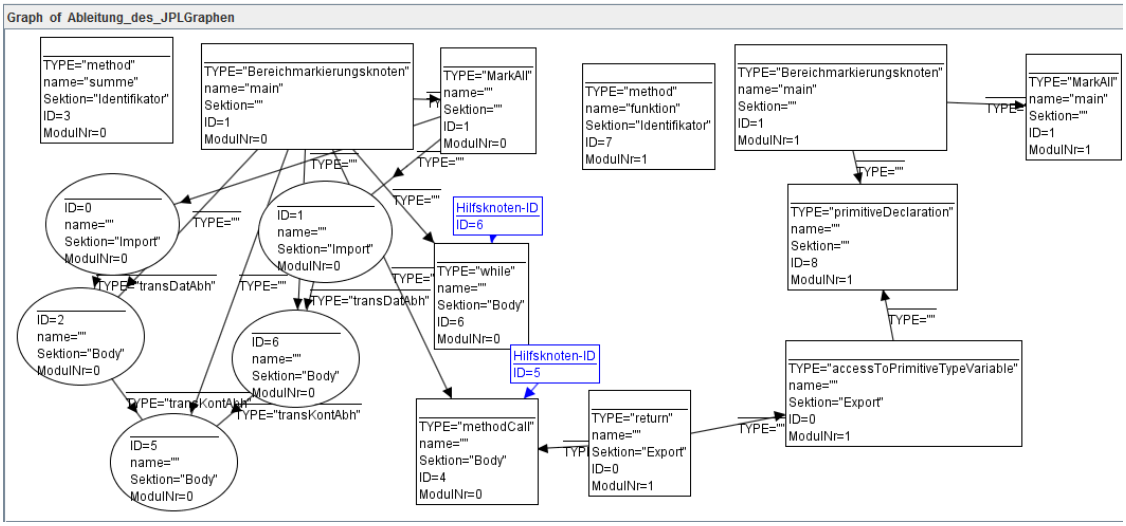


Abbildung 72: Aus dem JPL-Graphen abgeleitete Suchmustervariante

Der oben gezeigte Graph wird in einer Datei gespeichert. Diese Datei findet sich in Abbildung 70 im Kasten *Implementierungsvarianten, Dateien (j0-jx)*, welcher über die Kante *Gesamtmustersuche* referenziert wird.

Folgende Screenshots zeigen die Datei, welche zur Mustersuche verwendet wird. Die Regeln wurden in die Datei manuell eingefügt. Die Datei wird in Abbildung 70 im grau hinterlegten Kasten *Datei zur Mustersuche - pro Implementierungsvariante eine Regel* beschrieben.

Die Regelsätze wurden farbig umrahmt, um zu zeigen, über welchen Schritt diese erstellt wurden. Im Anhang Kapitel 11.3 Tabelle 7 in der Spalte *Regel einfügen* wird für jede einzelne Regel auf die Prozessschritte referenziert, durch welche die jeweilige Regel eingefügt wird. Die Regelsätze ergeben sich aus dem Graphen der Suchmustervariante in Abbildung 72.

Regelsätze:

- Gelber Rahmen: In diesem Regelsatz sind technische Hilfsregeln enthalten die über den AJSDG und JCG ausgeführt werden müssen, damit diese vollständig erstellt werden. Die Generierung dieser Regeln ist im Block *Erstellung der technischen Hilfsregeln.* beschrieben.
- Grüner Rahmen: Über diese Regeln werden die Bereiche markiert, in welchen die Muster der einzelnen Module gesucht werden. Diese werden über den Block: *Erstellung der Regeln zur Bereichsmarkierung für Methoden, Schleifen und Bedingungen.* generiert.
- Blauer Rahmen: Die Regeln markieren Bereiche außerhalb der zuvor für die Module markierten Gültigkeitsbereiche. Die Regeln werden über den Block: *Erstellung der Regeln zur Markierung von Elementen außerhalb der Gültigkeitsbereiche.* generiert. In Abbildung 70 werden die grün und blau umrahmten Regeln im Kasten *Erstellung der Regeln, welche die Bereiche markieren, innerhalb derer das Muster gesucht wird (Schritte 7– 22).* erfasst.
- Oranger Rahmen: Die Regeln generieren die transitive Hülle über dem AJSDG. Die Regeln werden über den Block *Erstellung der Regeln zur Generierung der*

Transitiven Hülle und des Pfades. eingefügt. In Abbildung 70 werden die orange umrahmten Regeln im Kasten *Erstellung der Regeln, welche die Transitive Hülle und die Pfadstrukturen generieren falls notwendig (Schritte 23 – 27).* erfasst.

- Brauner Rahmen: Über diese Regel wird die Mustersuche durchgeführt. Die Regel wird über den Block *Erstellung der Regeln zur Identifikation des Musters im Quellcode, Variante vollständige Mustersuche.* eingefügt. In Abbildung 70 wird die braun umrahmte Regel im Kasten *Erstellung der Regeln zur Gesamtmustersuche aus j0 – jx und einfügen in Datei (Schritte 28a – 31a).* erfasst.

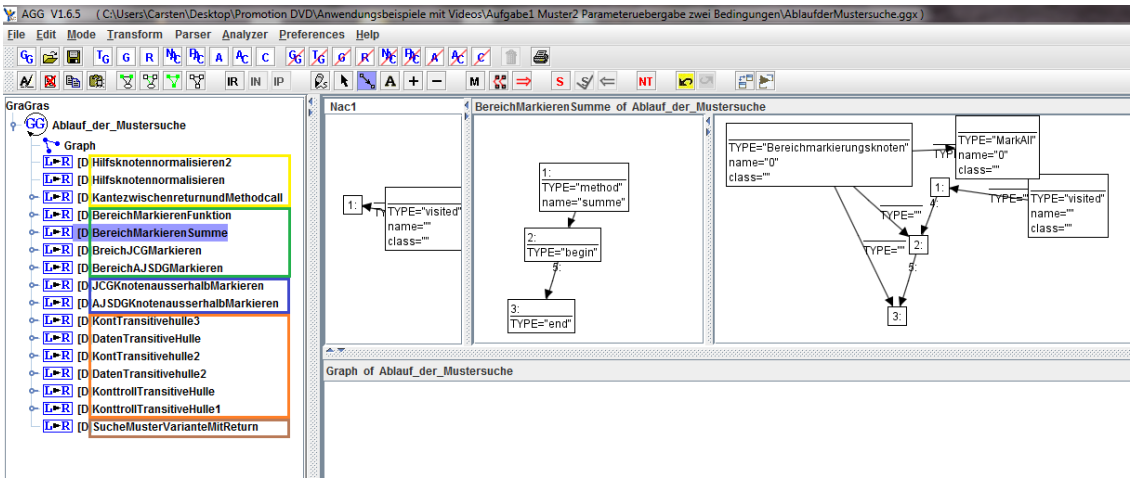


Abbildung 73: Regelsätze zur Gesamtmustersuche, gezeigt: Regel zur Bereichsmarkierung

In Abbildung 73 wird die Regel zur Markierung des Gültigkeitsbereichs der Methode *summe* dargestellt. Die Attribute *TYPE* und *name* des Knoten 1 auf der linken Regelseite und das Attribut *name* des Knotens vom Typ *Bereichsmarkierungsknoten* auf der rechten Regelseite werden im Rahmen der Regelerstellung parametrisiert.

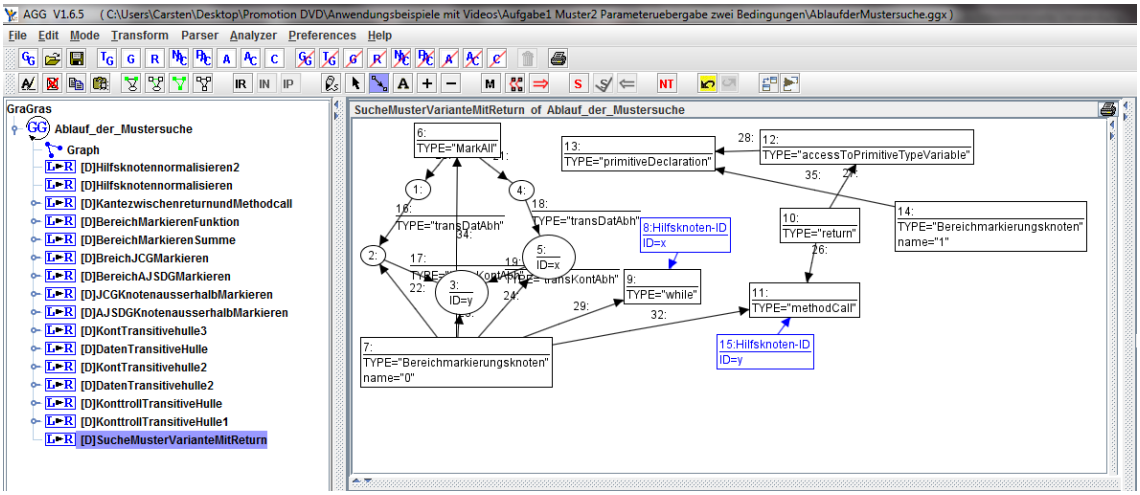


Abbildung 74: Regel zur Suche des Gesamtmusters

Abbildung 74 identifiziert das Gesamtmuster im zu analysierenden Quellcode. Die Regel wird aus dem abgeleiteten JPL-Graphen heraus generiert. Ein Quelltextsegement, das über diese Regel identifiziert wird, wurde über Abbildung 72 beispielhaft angegeben.

Über die oben gezeigte Datei kann nun automatisiert die Suchmustervariante in einer Graphrepräsentation des zu analysierenden Quellcodes gesucht werden. Die Syntaxelemente sind in Kapitel 3. aufgeführt.

Beispiel zur sektionsbasierten Mustersuche

Die Anwendung des zuvor beschriebenen Ableitungsprozesses für die Variante der sektionsbasierten Mustersuche wird im Folgenden für das Anwendungsbeispiel *Aufgabe4 Muster5 Lokale Variable* dargestellt. Dieses Beispiel wird in Kapitel 8.4 beschrieben und der Aufgabenkontext erläutert.

Der Screenshot unten zeigt den JPL-Graphen, welcher in der Graphsektion von AGG beschrieben wurde. Ausgehend von diesem abstrakten JPL-Graphen wird über Ableitungsregeln eine Suchmustervariante generiert, welche nach einer konkreten Implementierungsvariante des Musters sucht. Die Regeln wurden für dieses Beispiel manuell erstellt.

Hinweis: Das Attribut *ID* dient nur zur eindeutigen Identifizierung der im JPL-Graphen spezifizierten Elemente, um die Zuordnung von AJSDG- zu JCG-Knoten zu spezifizieren. Die für den Ableitungsprozess (Kopieren von Teilgraphen in Regeln) notwendige Identifizierung der Elemente erfolgt über die von AGG automatisch generierten IDs, welche in der GGX-Datei hinterlegt sind.

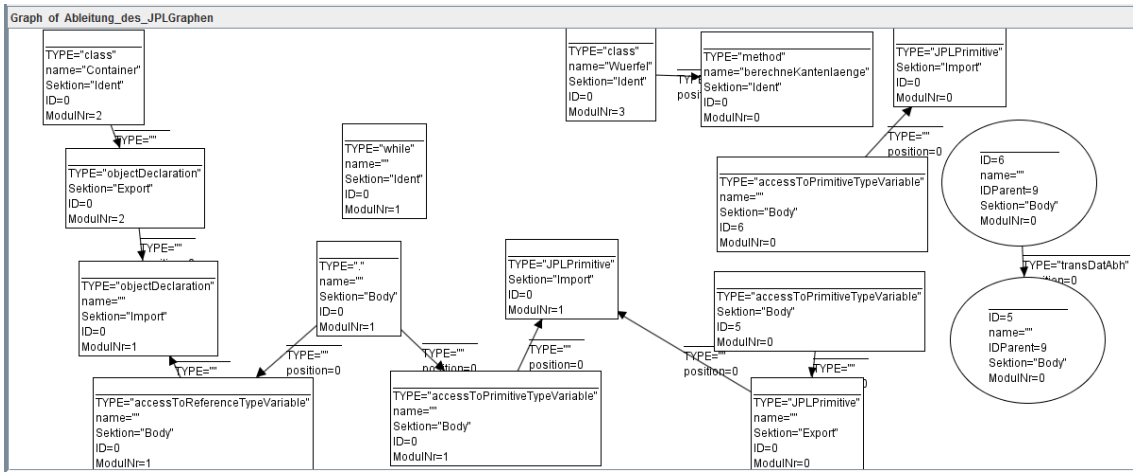


Abbildung 75: JPL-Graph des Suchmusters

Der folgende Screenshot zeigt die Suchmustervariante, welche nach dem automatisierten Ablauf der Ableitungsregeln für dieses Beispiel entsteht. Dieser Vorgang wird in Block *Ableitung des abstrakten JPL-Graphen in Graphvarianten, welche die zu suchenden Implementierungsvarianten abdecken.* beschrieben. In Abbildung 70 ist dies im Kasten *Ableitung des JPL-Graphen in mehrere Graphvarianten, welche die zu suchenden Implementierungsvarianten abdecken (Schritte 1 – 6). Unterscheidung der Suchvariante in Schritt 5.* aufgeführt.

In dieser Variante wird die Kantenlänge von der Methode *berechneGesamtkantenlaenge* an die *while*-Schleife durch eine *return*-Anweisung übergeben (Ersetzung der *JPLPrimitive*-Schnittstellenstruktur). Weiterhin wird die Übergabe mittels *objectDecaration* (auf der linken Seite des Graphen) aufgelöst und der *JPLPrimitive*-Knoten rechts oben im Graphen durch den Zugriff auf eine Membervariable ersetzt. Zu beachten ist, dass das Attribut *Sektion* der neu erstellten Knoten nur gesetzt wird, falls dies für die weitere Bearbeitung notwendig ist. Weiterhin wurde die Modulzuordnung der Knoten (zu erkennen am Attribut *ModulNr)* über Beziehungen zu *Bereichsmarkierungsknoten* dargestellt.

Über diese Variante wird z.B. folgendes Quellcodesegment identifiziert:

```
class ...{
double kantenlaenge;
}
class Wuerfel ... {
                    public double berechneKantenlaenge() {
                    double back;
                          back=12*kantenlaenge;
                          return back;}
}
public class Container {
        Wuerfel obj;
                  while ... {
                      obj.berechneKantenlaenge();
                  }
}
```

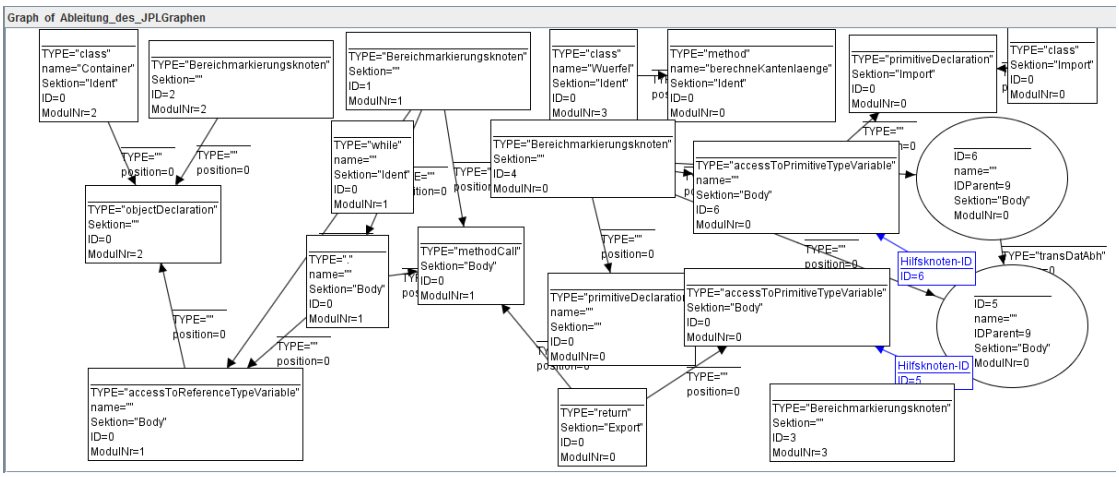


Abbildung 76: Aus dem JPL-Graphen abgeleitete Suchmustervariante

Der oben gezeigte Graph wird in einer Datei gespeichert. Diese Datei findet sich in Abbildung 70 im Kasten *Implementierungsvarianten mit Sektionsmarkierungen (Dateien j0-jx)*, welcher über die Kante *Sektionsbasierte Mustersuche* referenziert wird. Die Sektionen wurden hier noch nicht markiert.

Die folgenden Screenshots zeigen die Datei, welche zur Mustersuche verwendet wird. Die Regeln wurden in die Datei manuell eingefügt. Die Datei wird in Abbildung 70 im grau hinterlegten Kasten *Datei zur Mustersuche - pro Implementierungsvariante ein Regelsatz* aufgeführt.

Die Regelsätze wurden farbig umrahmt um zu zeigen, über welchen Schritt diese erstellt wurden. Die Anwendung des Prozesses wird in Kapitel 11.3. Tabelle 16 in der Spalte *Regel einfügen* für jede Regel einzeln dargestellt, indem auf die Prozessschritte referenziert wird, durch welche die jeweilige Regel eingefügt wird. Die Regelsätze ergeben sich aus dem Graphen der Suchmustervariante in Abbildung 76.

Regelsätze:

- Gelber Rahmen: In diesem Regelsatz sind technische Hilfsregeln enthalten die über den AJSDG und JCG ausgeführt werden müssen, damit diese vollständig erstellt werden. Die Generierung dieser Regeln ist im Block *Erstellung der technischen Hilfsregeln.* beschrieben.
- Dunkelgrüner Rahmen: Über diese Regeln werden die Bereiche markiert, in welchen die Muster der einzelnen Module gesucht werden. Diese Regeln werden über den Block: *Erstellung der Regeln zur Bereichsmarkierung für Methoden, Schleifen und Bedingungen.* und die Anweisung Nummer 15 generiert.
- Hellgrüner Rahmen: Über diese Regeln werden die hierarchisch abhängigen Bereiche markiert, in welchen die Muster der einzelnen Module gesucht werden. Diese Regeln werden über den Block: *Erstellung der Regeln zur Bereichsmarkierung für hierarchisch geschachtelte Bereiche.* generiert.
- Oranger Rahmen: Die Regeln erstellen die transitive Hülle über dem AJSDG. Die Regeln werden über den Block *Erstellung der Regeln zur Generierung der Transitiven Hülle und des Pfades.* eingefügt. In Abbildung 70 werden die orange umrahmten Regeln im Kasten *Erstellung der Regeln, welche die Transitive Hülle und die Pfadstrukturen generieren falls notwendig (Schritte 23 – 27).* erfasst. Es ist hier nur eine Regel angegeben, da nur nach direkten Datenabhängigkeiten gesucht wird. Dies ist bedingt durch die manuelle Nachbearbeitung des SDGs für dieses Beispiel, wodurch die Suche nach indirekten Datenabhängigkeiten nicht mehr notwendig ist.
- Brauner Rahmen: Über diese Regeln wird die Mustersuche durchgeführt. Die Regeln werden über den Block *Erstellung der Regeln zur Identifikation des Musters im Quellcode, Variante sektionsbasierte Mustersuche.* eingefügt. In Abbildung 70 wird der braun umrahmte Regelsatz im Kasten *Erstellung der Regeln zur sektionenbasierten Suche aus j0 – jx und einfügen in Datei. Übergabestrukturen werden abgeleitet aus dem Gesamtmuster (Schritte 28b – 31b)* erfasst.

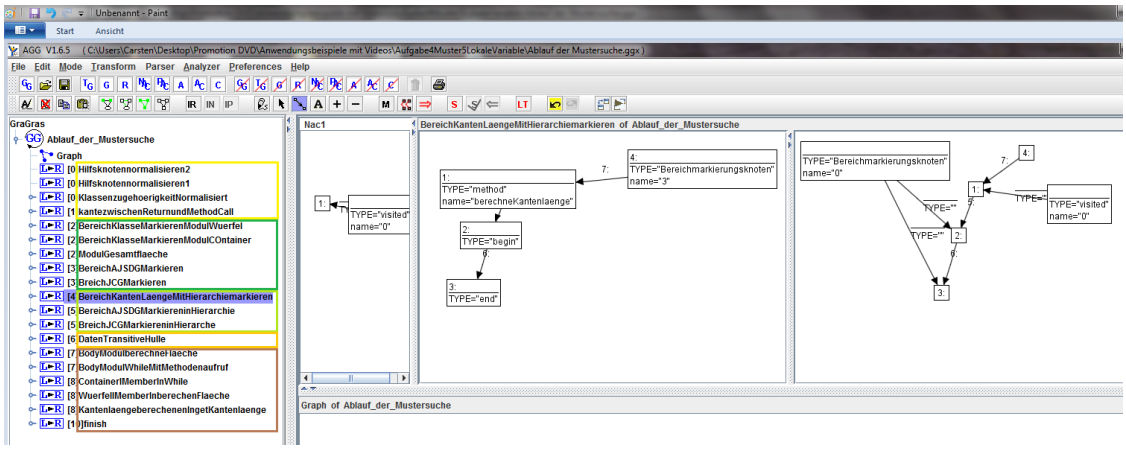


Abbildung 77: Regelsätze zur sektionsbasierten Mustersuche, gezeigt: Regel zur Bereichsmarkierung

In Abbildung 77 wird die Regel zur Markierung des Gültigkeitsbereichs der Methode *berechneKantenlaenge*, die der Klasse Würfel hierarchisch unterstellt ist, dargestellt. Die Attribute *TYPE* und *name* des Knotens 1 und das Attribut *name* des Knotens 4 (ModulNr des Moduls, das die Klasse Würfel erfasst) auf der linken Regelseite, sowie das Attribut *name* des Knotens vom Typ *Bereichsmarkierungsknoten* auf der rechten Regelseite (ModulNr des Moduls, das die Methode erfasst) werden im Rahmen der Regelerstellung parametrisiert.

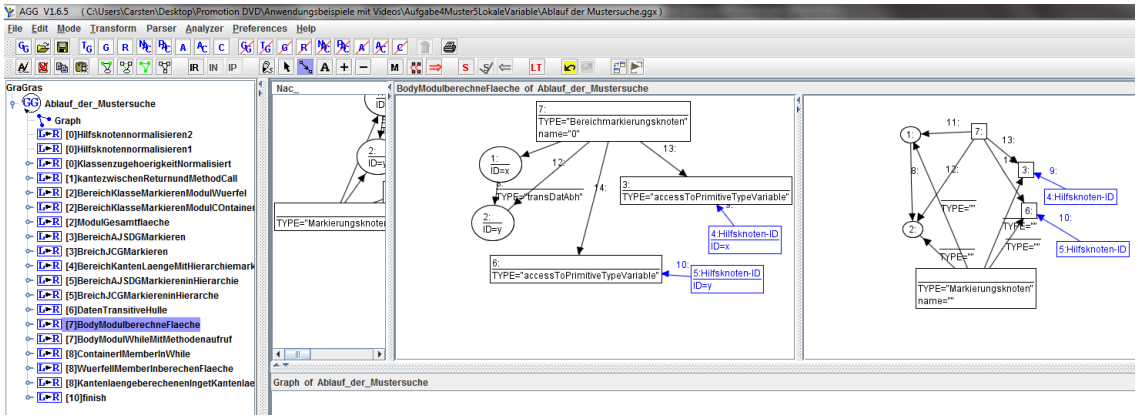


Abbildung 78: Regel zur Suche nach einer Körperstruktur

In Abbildung 78 ist die Regel zur Suche und Markierung des Musters dargestellt, welches in der Körpersektion des Moduls spezifiziert ist, das den Gültigkeitsbereich der Methode *berechneKantenlaenge* umfasst. Das Muster besteht aus JCG-Elementen, die in Anweisungen, welche durch die aufgeführten AJSDG-Elemente repräsentiert werden, enthalten sind. Die Elemente können in Abbildung 76 über die Attributbelegungen *ModulNr=0* und *Sektion= Body* identifiziert werden. Die Kante von Knoten 7 zu Knoten 6 wird im Laufe des Ableitungsprozesses erstellt.

Über diese Regel wird z.B. folgendes Quellcodesegment identifiziert:

```
public double berechneKantenlaenge() {
        back=12*kantenlaenge;
        return back }
```

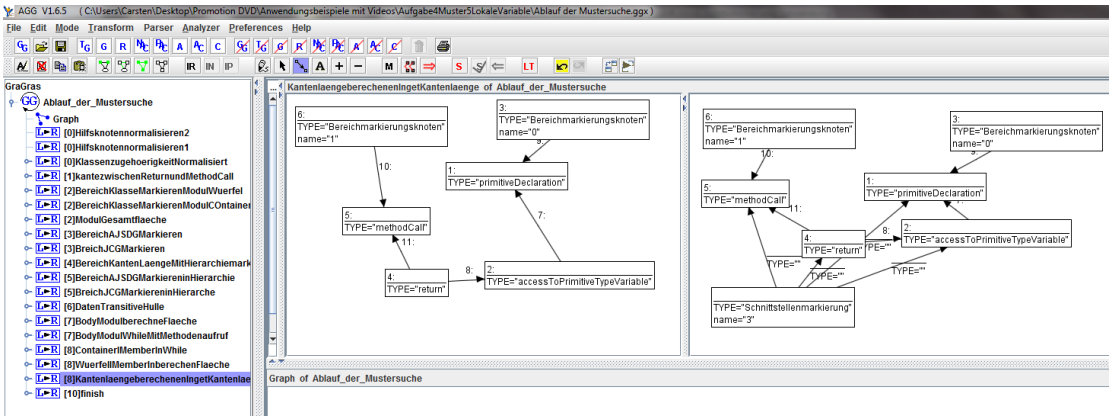


Abbildung 79: Regel zur Suche nach einer Schnittstellenstruktur

In Abbildung 79 ist die Regel zur Suche und Markierung der Schnittstellenstruktur, welche die Übergabe eines Wertes aus der Methode *berechneKantenlaenge* in eine *while*-Schleife erfasst, dargestellt. Die Schnittstellenstruktur wurde im abgeleiteten JPL-Graph (s. Abbildung 76) erkannt und in die Datei zur Mustersuche als Regel übertragen.

Über diese Regel wird z.B. folgendes Quellcodesegment identifiziert:

```
public double berechneKantenlaenge() {
   double back;
   return back;}
         }
...
   while ...{
         berechneKantenlaenge();
  }
```

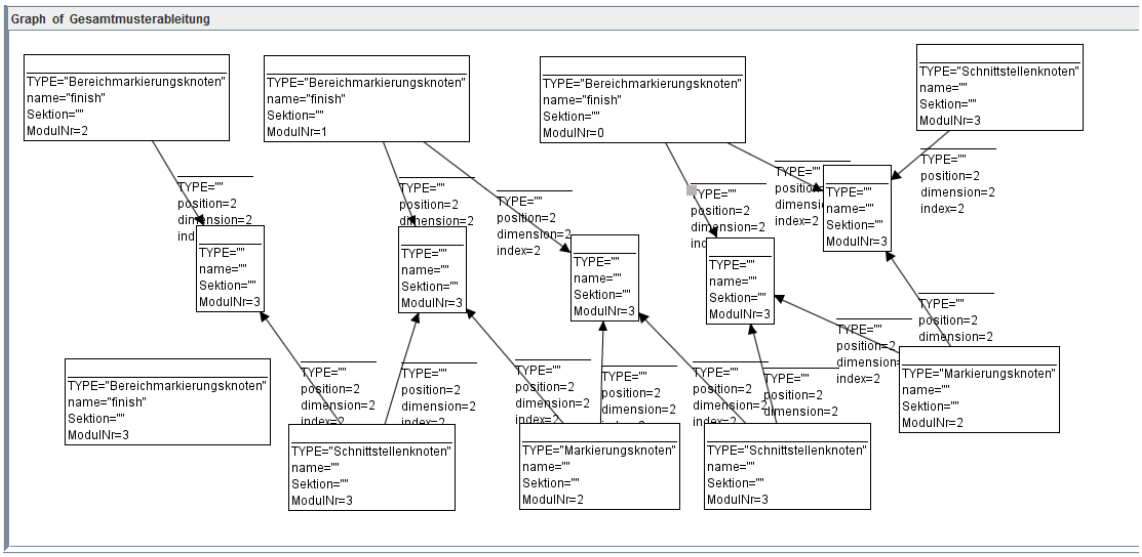


Abbildung 80: Aus dem JPL-Graphen abgeleiteter Gesamtgraph (rechts)

In Abbildung 80 wird der Graph gezeigt, welcher die verschiedenen Markierungen der vorhergehenden Regeln erfasst und das Gesamtmuster sucht (Regel *final* im braun umrahmten Regelsatz in Abbildung 77). Das Suchmuster wird ausgehend vom JPL-

Graphen, welcher in Abbildung 75 dargestellt wird, durch Transformationsregeln automatisiert abgeleitet. Das Einfügen dieses Musters in den Regelsatz zur Mustersuche erfolgte nachfolgend manuell. Ein Quelltextsegement, das über diese Regel identifiziert wird, wurde über Abbildung 76 beispielhaft angegeben.

Anmerkung zum Stand der Automatisierung des Ableitungsprozesses:

Das Ziel der Implementierung der verschiedenen Regelsätze und Prozessschritte, die im Rahmen der Anwendungsbeispiele aus Kapitel 8. erfolgte, ist es zu zeigen, dass der Ansatz sich praktisch realisieren lässt, d.h. dass Suchmustervarianten automatisch über einen Regelablauf in einem Graphtransformationstool erstellt werden können und der automatisierte Ablauf der Regeln, welche sich aus dem Muster ergeben, die zu suchenden Graphen in der Quelltextrepräsentation findet. Nicht im Fokus steht die technische Übertragung des Graphen der Mustervariante in die Graphregeln, welche die Mustersuche realisieren.

Funktionen zur technischen Übertragung von Teilstrukturen des abgeleiteten JPL-Graphen in den Regelsatz zur Mustersuche:

- Kopieren von Attributen in ein bestehenden Regeltemplate (Block: Bereiche markieren)
- Kopieren bestehender Regeln zur Erstellung der transitiven Hülle (Block: Generierung der Transitiven Hülle)
- Kopieren einer Graphstruktur in die linke und rechte Seite einer neu erstellten Regel (Kopieren von Graphen zur Suche von Mustern die sich auf die Körpersektion, die Variablenübergabe und das Gesamtmuster beziehen)

Falls in künftigen Arbeiten die vollständige Toolkette realisiert werden soll, muss vorher der Einsatz alternativer Tools geprüft werden. Die bestehende Implementierung basiert auf dem Tool AGG. Aktuell wird angenommen, dass bei einer Weiterverfolgung der Implementierung auf das Projekt Henshin [Hen14] und die hier verwendete Toolunterstützung umgeschwenkt werden sollte, da hier auf neuere Technologien (EMF) und eine breitere Community gebaut werden kann. Dies ist ein weiterer Grund aus dem die AGG- bzw. ggx- (XML-Format von AGG, in dem die Graphgrammatiken gespeichert werden) spezifische Implementierung nicht weitergeführt wurde.

In den vorhergenden zwei Beispielen wurde gezeigt, wie aus einem JPL-Graphen die Datei zur Mustersuche erstellt wird. Die Mustersuche kann nachfolgend automatisiert auf einem Server ausgeführt werden. Die hierzu eingesetzte Umgebung wird im folgenden Kapitel erläutert.

7.3. Automatisierter Ablauf der Mustersuche

In diesem Kapitel wird der Ablauf der Mustersuche beschrieben, nachdem die Suchmuster spezifiziert, abgeleitet und die resultierenden Graphregeln auf den Server geladen wurden.

Ablauf zur Mustersuche auf dem Server

Nachdem der zu überprüfende Quellcode an den Server übertragen wurde, wird dieser in einer Datenbank gespeichert. Hiernach wird der Code durch das Tool *Java2GGX* in eine JCG-Graphstruktur transformiert. Nachfolgend werden auf diese Graphstruktur die für diese Übung definierten Regeln/Suchmuster angewendet, wobei die Abfolge der Regeln durch das Tool *RuleControl* und die Anwendung der Regeln über das Tool *AGG* [agg14] realisiert werden. Sobald ein Fehler erkannt wird, wird ein Ergebnisknoten generiert, welcher die Fehlerbeschreibung enthält. Wird ein Muster gefunden, so wird ein Ergebnisknoten generiert, welcher die Gültigkeitsbereiche der Variablen beschreibt, welche in dem Muster importiert oder exportiert werden. Dies dient zur Unterstützung einer möglichen nachfolgenden manuellen Überprüfung der automatisiert erkannten Analyseergebnisse, um *false positives* auszuschließen. Nach dem Ablauf aller Überprüfungsregeln werden die erzeugten Ergebnisknoten geparsed und die Meldungen in der Datenbank dem hier bereits gespeicherten zugehörigen Quellcode zugeordnet. Sobald der Übungsleiter die Evaluierung der Lösung abruft, wird ihm ein Report angezeigt, welcher die entsprechenden Meldungen enthält. Nach dessen Freigabe ist dieser Report auch für den Studenten einzusehen.

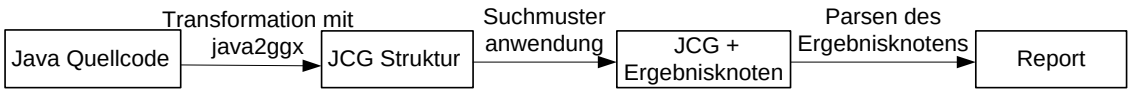


Abbildung 81: Workflow auf dem Server zur automatisierten Prüfung von Übungsaufgaben.

Ergänzung: Falls das Suchmuster AJSDG Elemente enthält werden die im Rahmen von [WH07] erstellten Regeln auf der JCG Struktur ausgeführt und auf der sich hierdurch ergebenden Datei das Suchmuster angewendet. Dieser Schritt wurde für die Überprüfung der in Kapitel 8 dargestellten Anwendungsbeispiele ausgeführt. Der sich ergebende AJSDG Graph muss im aktuellen Implementierungsstand manuell nachkorrigiert werden.

Im Weiteren werden die Tools beschrieben, welche für die Realisierung der Testumgebung genutzt werden:

Attributed Graph Grammar System [agg14]: Das System unterstützt die Entwicklung und Ausführung von attributierten Graph Grammatiken. Hierzu stellt AGG eine Oberfläche zur Verfügung, über welche die entsprechenden Regeln spezifiziert und ausgeführt werden können. Weitere Features dieses Tools sind die Möglichkeit Typgraphen zu erstellen, die Unterstützung der Critical Pair Analysis, sowie die Bereitstellung eines Layer basierten Ansatzes zur Steuerung von Regelabfolgen. Es kann sowohl der SPO-, als auch der DPO- Ansatz verwendet werden. Zusätzlich zur Nutzung der AGG über diese GUI, wird eine API zur Verfügung gestellt, welche u.a. die Definition und Ausführung von Graphregeln ermöglicht.

Java Compiler Compiler und Java Tree Builder: Der ursprünglich von SUN entwickelte Java Compiler Compiler (JCC) erstellt mittels einer gegebenen Java-Grammatik ein Programm, welches diese Grammatik in gegebenen Java Applikationen untersuchen, bzw. Übereinstimmungen mit dieser Grammatik finden kann. Zusätzlich zu diesem Parser wird mittels des Java Tree Builders die Möglichkeit bereitgestellt, einen Abstrakten Syntax Baum aus dem gegebenen Programm zu erstellen. Die Tools, Anleitungen zu deren Verwendung und die entsprechenden Grammatiken werden auf der Home Page des Projekts bereitgestellt [JCC14].

Zur Realisierung der Testumgebung wurden am Lehrstuhl „Specification of Software Systems“ an der Universität Duisburg-Essen verschiedene Tools erstellt. Diese werden im Folgenden beschrieben:

Java2GGX: Dieses Tool transformiert Java Source Code in einen Abstrakten Syntax Graph. Dieser Graph wird im GGX Format abgelegt, so dass er nachfolgend im AGG-Tool weiterverwendet werden kann. Der Code wird mittels des Parser-Generators JavaCC, welcher das Tool Java TreeBuilder enthält, analysiert und der AST wird erstellt. Nachfolgend erfolgt die Erweiterung des AST zum JCG, indem implizite Abhängigkeiten explizit über Kanten dargestellt werden. Durch diese Erweiterung wird die Spezifikation von Graphregeln (Suchmustern) vereinfacht, welche auf diesem Graphen aufsetzen.

Rule Control: Über das Tool RuleControl kann die Abfolge festgelegt werden, über welche Graphregeln ausgeführt werden. Hierbei sind sowohl einfache Sequenzen möglich, als auch die Definition von Bedingungen im Regelablauf, auf Grund derer entschieden wird, welche Regelfolge angewendet wird. Die Ablaufsequenz wird in einem Textfile abgelegt. Zur Ansprache der entsprechenden Regeln wird die API des AGG Tools genutzt.

Jack: Im Java Checker [GSB08] wurden die zuvor beschriebenen Tools zusammengefasst und der in Kapitel 7.1 beschriebene Workflow realisiert. Der Fokus liegt hier auf der Prüfung von Testaten und Übungsaufgaben mittels Mustersuche auf dem JCG der Lösungen, wobei die Integration des AJSDG noch aussteht. Die Testumgebung wurde in den vergangenen drei Jahren zur Prüfung hunderter Übungsaufgaben eingesetzt, welche im Rahmen des Kurses „Grundlagen der Informatik“ durchgeführt wurden. Jack wurde als Web-Service implementiert, so dass die Studenten ihre Lösungen über einen Browser hochladen können, woraufhin die Prüfung automatisch angestoßen wird. Die Ergebnisse, sowie die vom Studenten hochgeladene Übungslösung werden dem Prüfungsleiter daraufhin in einem Web-Portal dargestellt. Somit ist es möglich manuell einzelne Analyseergebnisse nachzuprüfen, falls dies notwendig erscheint.

7.4. Zusammenfassung

Wird der Prozess von der Erstellung eines JPL-Musters bis zur Ausführung der Mustersuche und Ausgabe des Suchergebnisses betrachtet (s. Abbildung 27: Prozess der Ableitung des JPL-Graphen in Graphtransformationsregeln sowie Ausführung der Mustersuche), so wurde in Kapitel 7.2. ein Grobdesign zur toolbasierten Unterstützung von der Erstellung eines JPL-Musters und nachfolgend der automatisierten Ableitung des Musters in die verschiedenen Suchmustervarianten, bis zur Erstellung des Regelsatzes zur Mustersuche, beschrieben. Die automatisierte Ausführung der Mustersuche, d.h. der Regeln des Regelsatzes, und die Erstellung des Ergebnisknotens erfolgt über die in Kapitel 7.3. gezeigte Testumgebung, welche an der Universität Duisburg-Essen bereits über mehrere Jahre praktisch eingesetzt wird.

Die in diesem Kapitel vorgestellte Toolumgebung wurde realisiert, um die Überprüfung von Java Übungsaufgaben zu unterstützen, welche z.B. in Java Grundlagenvorlesungen standardmäßig durchgeführt werden. Da diese Übungen von einer großen Anzahl Studenten besucht werden, ist eine automatisierte Unterstützung hier unumgänglich, um die Vielzahl an Lösungen effizient korrigieren zu können. Die Effektivität dieses Ansatzes und der durch Jack realisierten Umgebung hat sich bei der Überprüfung tausender Aufgabenlösungen erwiesen. Einige dieser Übungen werden im folgenden Kapitel näher betrachtet.

8. Anwendungsbeispiele

In diesem Kapitel wird der Einsatz von JPL-Mustern im Kontext verschiedener Java-Übungsaufgaben betrachtet. Die Aufgaben wurden in ähnlicher Form in der Übung zur Vorlesung „Programmierung“ an der Universität Duisburg-Essen eingesetzt und repräsentieren somit typische Java-Übungen, welche im Rahmen von Lehrveranstaltungen genutzt werden. Zur automatisierten statischen Überprüfung der studentischen Lösungen wurden von verschiedenen Übungsleitern Suchmuster basierend auf dem AST spezifiziert, welche über Graphregeln ausgeführt wurden.

Im Folgenden werden nicht die vollständigen Übungsaufgaben betrachtet, sondern die Teilbereiche, welche die Demonstration verschiedener Aspekte der JPL-Muster ermöglichen. Zu diesem Zweck werden verschiedene komplexe Suchmuster dargestellt und aufgezeigt, wie sich diese von der bisher verwendeten Mustererstellungstechnik mittels reiner AST-Strukturen abheben. Der Fokus liegt sowohl auf der modularisierten Repräsentation der Suchmuster als auch auf der kombinierten Verwendung von AJSDG- und JCG-Strukturen. Für jede Übungsaufgabe wird nachfolgend analysiert, inwieweit die bisher erstellten AST-Muster durch die neuen JPL-Muster ergänzt werden, bzw. bei welchen Aspekten die Spezifikation der Muster vereinfacht wird. Weiterhin wird gezeigt, dass der in dieser Arbeit dargestellte Prozess von der JPL-Musterspezifikation bis zur Erstellung der Ergebnisknoten über ein Graphtransformationstool technisch realisierbar ist.
Es wird beschrieben, inwieweit die Aufgabenstellung freier gestellt werden kann, als dies in bisherigen Aufgaben möglich war, ohne dass sich der Aufwand der Musterspezifikation und Mustersuche signifikant erhöht. Um dies zu demonstrieren, wurden sowohl die Übungsaufgaben als auch die verwendeten Studentenlösungen dahingehend abgeändert, dass diese eine höhere Anzahl an Implementierungsvarianten zulassen, als die Originalaufgaben und somit auch die ursprünglichen Übungslösungen besaßen. Änderungen in den Aufgabenstellungen werden in den Beispielen hervorgehoben dargestellt.
Die Musterableitung und die Durchführung der Suche zu den im Folgenden betrachteten JPL-Mustern wurde beispielhaft im Graphtransformationstool AGG realisiert. Es wurden fünf Suchmuster über das Tool AGG als JPL-Graph spezifiziert. Diese wurden jeweils auf drei von Studenten erstellten unterschiedlichen Lösungsvarianten angewendet, so dass insgesamt 15 Übungslösungen überprüft wurden. Die in den nächsten Kapiteln dargestellten Beispielcodefragmente stammen aus diesen Lösungen. Die Ableitung der JPL-Muster erfolgte zum Teil automatisiert über Graphregeln und teilweise manuell, d.h. Suchmuster wurden manuell in die Ablaufdatei zur Mustersuche eingetragen.

Folgende Übungsaufgaben werden in diesem Kapitel betrachtet:

1. Berechnung einer mathematischen Funktion
2. Lager als Liste
3. Geometrische Objekte mit Fokus auf Vererbungsstrukturen
4. Verwaltung geometrischer Objekte als Liste

Die Aufgaben sind aus Testataufgaben und aus komplexeren Miniprojekten entnommen. Der Fokus der JPL-Muster liegt auf der Darstellung komplexer

Zusammenhänge über Gültigkeitsbereiche hinweg. Es werden sowohl ein einzelnes Suchmuster als auch miteinander verbundene komplexe Muster beschrieben. Die Suchmuster der ersten drei Beispiele zeigen jeweils verschiedene Einzelaspekte der JPL auf, während diese in der letzten Übungsaufgabe zusammengeführt werden. Abschließend werden Kriterien für geeignete Übungsaufgaben, deren Lösungen durch JPL-Muster überprüft werden können, aufgeführt. Weiterhin werden Hinweise zur Erstellung von JPL-Mustern aufgeführt. Die Kriterien und Hinweise haben sich aus der Konzeption und Implementierung der im Folgenden dargestellten Suchmuster ergeben.

8.1. Mathematische Funktion

Aufgabenstellung:

Programmieren Sie die Methode *funktion*. Sie soll für ein gegebenes x den Funktionswert von f(x) zurückgeben, für folgende Funktion:

*f(x)=x*0,5 //Halbierung des Wertes*

Programmieren Sie mit Schleifen:
Die Methode *summe:* Sie soll für ganze Zahlen in einem gegebenen Intervall die Summe der Funktionswerte dieser Zahlen errechnen und zurückgeben. Die Funktionswerte werden durch die Methode *funktion* berechnet. **Die Intervallgrenzen müssen größer als Null sein.**

Nutzen Sie zur Realisierung folgendes Pseudocodefragment:

public class Miniprojekt1 {
Erstelle eine Methode funktion mit einer Ergebnisrückgabe vom Typ double**; ursprüngliches konkretes Fragment: public double funktion {int x}, **Es wird dem Studenten überlassen, wie er das Funktionsergebnis zurückgibt.
*{ **//** //hier die Lösung eintragen...*
}

Erstelle eine Methode summe mit dem Rückgabetyp double**; ursprüngliches konkretes Fragment:public double summe(int von, int bis), **Es wird dem Studenten überlassen, wie er das Summenergebnis zurückgibt. Es wird dem Studenten überlassen, wie er die Intervallgrenzen übergibt.
{
//einschließlich von außen gegebener Grenzen
//hier die Lösung eintragen
}}

Einer der Gründe für die ursprünglich striktere Vorgabe der zu nutzenden Codestruktur ist die Einschränkung von Implementierungsvarianten der Aufgabenlösungen bezogen auf Wertübergaben in Gültigkeitsbereiche, um so nur eine kleine Anzahl von AST-Mustern zur statischen Überprüfung erstellen zu müssen. In dem vorgegebenen Codefragment mussten für eine korrekte Lösung Werte über Methodenparameter übergeben und über Parameter der *return*-Anweisung zurückgegeben werden. Bei Verwendung der JPL kann diese Einschränkung entfallen, da Varianten bezogen auf den Ex-/Import von Variablen in Gültigkeitsbereiche nicht explizit spezifiziert werden müssen.

Top Down-Analyse der funktionalen Anforderungen:

Die Lösungsstruktur kann ausgehend von den geforderten Klassen und Methoden initial in drei Module unterteilt werden; die Methoden *funktion*, *summe* und die Klasse *Miniprojekt1*.

Analyse der Einzelmodule:

1. In der Methode *summe* ist strukturell eine Schleife gefordert.
2. Die Methode *summe* muss ein Intervall betrachten. Das Intervall wird realisiert durch zwei von außen übergebene Variablen, welche jeweils eine Ober- und Untergrenze angeben, wobei die Untergrenze nicht unter Null liegen darf.
3. Die Schleife dieses Moduls ist abhängig von einer oder beiden Variablen.
4. Keine der Intervallvariablen darf kleiner als Null sein.
5. Innerhalb der Schleife müssen einer Variablen Werte von außerhalb der Methode zugewiesen werden.

6. In die Methode *funktion* wird ein Wert übergeben.
7. Der Wert muss in einer Berechnung genutzt werden.
8. Das Ergebnis muss zurückgegeben werden.

Analyse der Modulkombination:

9. Die Klasse *Miniprojekt1* muss die beiden Methoden enthalten.
10. Die Methode *funktion* wird von der Methode *summe* aufgerufen.
11. Das Ergebnis der Methode *funktion* wird in der Methode *summe* verwendet.

Im Folgenden wird ein JPL-Muster dargestellt, welches die Implementierung der oben definierten Anforderungen bezogen auf das Modul *summe* sucht. Es wird für dieses Suchmuster angenommen, dass in der Schleife beide Intervallgrenzen als Bedingungsparameter enthalten sind.

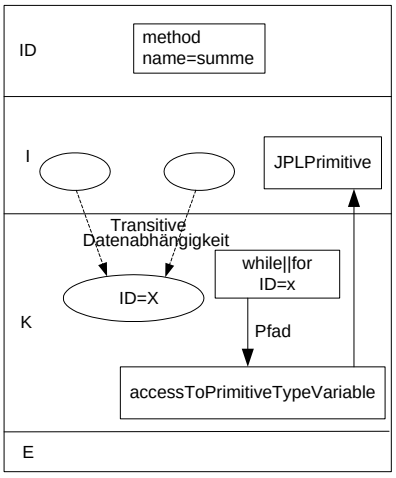


Abbildung 82: Mathematische Funktion, JPL-Muster 1

Über die transitive Datenabhängigkeitsstruktur wird ein Ausdruck gesucht, welcher von zwei importierten Variablenwerten direkt oder indirekt datenabhängig ist. Ziel ist die Identifikation sowohl der zu durchlaufenden Schleife als auch der Verwendung der

Intervallgrenzen im Schleifenkopf. Um zu erkennen, ob die beiden Variablen im Rahmen einer Schleifendefinition verwendet werden, wird das Muster um das entsprechende JCG-Element ergänzt (Anforderungen 1, 2, teilweise Anforderung 3). Es werden hier eine *while*-Schleife und eine *for*-Schleife erfasst. Der Pfad, welcher ausgehend vom Schleifenkopf auf eine Variablenreferenz verweist deren Wert importiert wird, gibt an, dass innerhalb der identifizierten *while*-Schleife ein Wert von außen gesetzt werden muss. Das Ziel ist, die Verwendung des Rückgabewerts der Methode *funktion* innerhalb der Schleife zu identifizieren, welche von den Intervallgrenzen abhängig ist. Die Belegung dieser Variablen in der Methode *funktion* wird hier nicht weiter betrachtet.

Codefragmente, die über dieses Muster erkannt werden:

```
classMiniprojekt1{
public double summe (int von, int bis){
        i=von;
        double sum=0;
        while (i>0 && i<=bis){
                … funktion(i);
                …
        } …

public double funktion (double x){
        return x…
}}

class Miniprojekt1{
int von;
int bis;
double global;
public double summe (){
        int i=von;
        double sum=0;
        while (i>0 && i<=bis)
                {…
                … global;
        …
        }…

public void funktion (int x){
        global=…
}}
```

Im ersten Code-Fragment werden die Intervallgrenzen als Methodenparameter übergeben. Die untere Grenze wird an eine Hilfsvariable zugewiesen, welche zusammen mit der Intervallobergrenze als Schleifenbedingung eingesetzt wird. Das Ergebnis der Methode *funktion* wird über einen *return*-Ausdruck übergeben. Im Gegensatz hierzu werden im zweiten Codefragment die Intervallgrenzen als Membervariablen implementiert und das Ergebnis der Methode *funktion* in einer globalen Variable gespeichert. Beide Varianten werden über das Muster erkannt.
Im Suchmuster wurde der Methodenaufruf selbst nicht spezifiziert. Falls diese Überprüfung explizit gewünscht ist, muss das Element *methodcall* hinzugefügt werden, oder ein entsprechendes separates Muster erstellt werden.

Während im oben dargestellten Muster für einen gültigen Match die Intervallgrenzen in einem Ausdruck referenziert werden müssen (Schleifenbedingung), wird im folgenden Muster die Implementierung von zwei Ausdrücken angenommen. Im Weiteren wird nicht das Pfadelement eingesetzt, sondern, als Alternativstruktur, die Kontrollabhängigkeit.

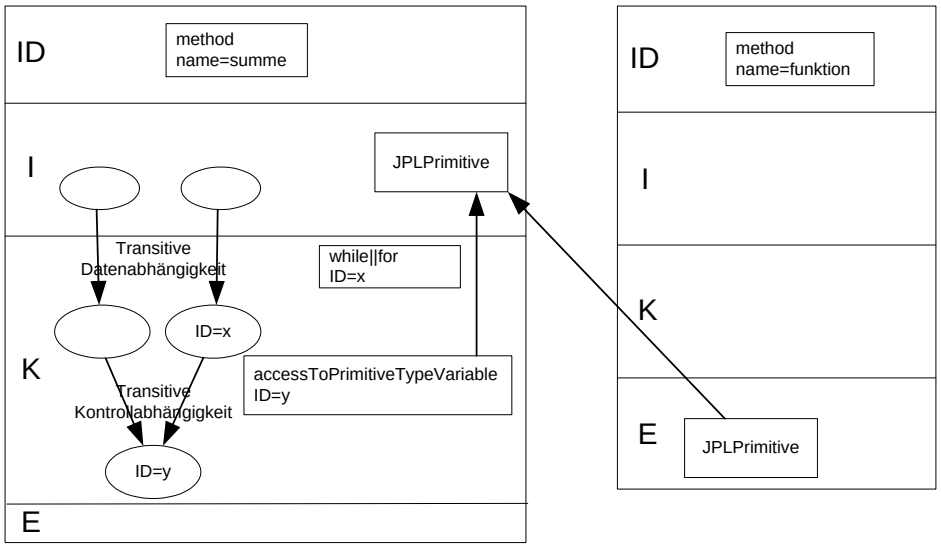


Abbildung 83: Mathematische Funktion, JPL-Muster 2

Im Gegensatz zu Muster 1 werden in diesem Muster Implementierungsvarianten erfasst, welche die Intervallgrenzen in zwei verschiedenen Bedingungen verwenden, von denen nachfolgend ein Ausdruck kontrollabhängig ist, welcher einen Wert aus der Methode *funktion* importiert. Die Verwendung von Hilfsvariablen wird über den Einsatz transitiver Datenabhängigkeitskanten berücksichtigt. Durch die transitiven Kontrollabhängigkeitskanten werden Ausdrücke unabhängig von der Art der Verschachtelung erkannt. So könnte z.B. zuerst eine *if*-Bedingung (z.B. für die Prüfung, ob die Intervalluntergrenze größer Null ist) und nachfolgend eine Schleife oder alternativ zuerst auch die Schleife implementiert werden. Es können vor dem importierenden Ausdruck auch noch weitere hier nicht betrachtete Bedingungen implementiert werden, ohne dass die Mustererkennung beeinträchtigt ist (Anforderungen 1, 2 und teilweise Anforderung 3). Weiterhin wird eine Werteübergabe von der Methode *funktion* in die Methode *summe* spezifiziert (Anforderung 11).

Beispiele für gültige Lösungen der Methode *summe*

```
public double summe(int von, int bis){
double sum;
if (von>0)) {
while (von<=bis){
                funktion(von);
                sum=sum+global;
                ...}}
```

oder

```
public double summe(int von, int bis){
double sum;
i=von;
if  (i>0){
while (i <=bis){
                    sum=sum+funktion(i);
                    ...}}
```

Vergleich JPL- zu AST-Mustern:

Während die Art der Werteübergabe zwischen *summe* und *funktion* im JPL-Muster durch eine JPLPrimitive-Knoten-Struktur abgebildet wird, müssten in reinen AST Mustern alle Implementierungsvarianten berücksichtigt werden. Da hier eine große Anzahl an Varianten möglich ist, wurde in der ursprünglichen Aufgabe nur die Übergabe als Methodenaufrufparameter betrachtet. Die Rückgabe des Funktionsergebnisses wurde nicht überprüft.
Weiterhin wurde in der ursprüngliche Aufgabenstellung für die Methode *summe* vorgegeben, dass die Intervallgrenzen als Parameter übergeben werden müssen. Durch die Verwendung von AJSDG-Knoten werden in den beschriebenen Mustern jegliche Datenabhängigkeiten betrachtet, welche die Ausdrücke im Gültigkeitsbereich der Methode nach außen hin besitzen. Somit werden auch Übergaben als Membervariablen der Klasse oder der Zugriff auf eine externe Klasse, welche die Intervallgrenzen verwaltet, erkannt. Im Gegensatz zur Verwendung der JPL-Schnittstellenknoten wird hier nur eine bestehende Abhängigkeit erkannt und nicht analysiert, welche Übergabemethode gewählt wurde.

Die Beziehungen zwischen den Variablen, welche das Intervall vorgeben, der geforderten Schleifenstruktur und der Verwendung der Methode *funktion* können über vielfältige AST- Strukturen realisiert werden. In obigem Beispielmuster wurde der geforderte Daten- und Kontrollfluss abgebildet, um über diese Abstraktionen die strukturell korrekte Lösung zu erfassen. In den Regeln zu der ursprünglichen Aufgabe wurde hier lediglich untersucht, ob in der Methode *summe* die Methode *funktion* aufgerufen wird. Die weiteren Beziehungen wurden aufgrund der hohen Implementierungsvariantenvielfalt nicht weiter betrachtet.

8.2. Lager als Liste

Aufgabenstellung:

Ein Unternehmen hat ein Lager, das nach dem FIFO (First in First out) Prinzip organisiert ist. Das Lager soll als einfach verkettete Liste dargestellt werden. Alle Elemente der Liste besitzen jeweils einen Namen von Typ *String*. Implementieren Sie für die Verwaltung nun folgende Methoden:

1. Eine Methode einfügen zur Erstellung der Lebensmittellisten, mit der Elemente in die (möglicherweise leere) Liste an deren Ende eingefügt werden können.
2. Eine Methode suchen, mit der Elemente anhand ihres Namens in den Lebensmittellisten gefunden werden können.
3. Es soll weiterhin eine Methode *entnehmen* erstellt werden, welche die Elemente nach dem FIFO Prinzip aus dem Lager entfernt.

Nutzen Sie für die Realisierung die folgende Vorlage:

```
public class Lebensmittelliste{
Lebensmittel fuss, kopf;
public void …einfuegen(Lebensmittel lm) {// Methode fuellen}
public ... suchen(String name) {// Methode fuellen}
}

public class Lebensmittel{
 String name;
... next;
public ...() {
// Parameterloser Konstruktor
// Bitte nicht entfernen.
… }}

public class Miniprojekt3 {
public static void main(String[] args) {
Lebensmmittlellist list=new Lebensmittelliste();
… }}
```

Es wird nicht vorgegeben, wo die Methode *entnehmen* implementiert werden soll. Sie kann in der Klasse des Codefragments oder z.B. einer eigenen Klasse implementiert werden.

Top Down Analyse der funktionalen Anforderungen:

Die Lösungsstruktur kann ausgehend von den geforderten Klassen und Methoden in fünf Module unterteilt werden; die Methoden*: einfügen*, *suchen*, *entnehmen*, die umfassende Klasse der Liste und die Klasse, welche die zu verwaltenden Waren enthält.

Ausgewählte Anforderungen an die Einzelmodule:

1. In der Methode *entnehmen* muss das Wurzelelement lesend verwendet werden.
2. In der Methode *suche* wird auf das Objekt *name* des zu verwaltenden Elementtyps zugegriffen.
3. Die Klasse Lebensmittelliste enthält als Membervariable die Wurzel der Liste.

4. Die Erstbelegung der Liste wird in einer Zeile implementiert. Dieser Zeile ist eine if-Bedingung vorgelagert.

Analyse der Modulkombination:

5. Die Methode *suche* ist hierarchisch der Klasse der Liste zugeordnet.
6. Die if-Bedingung muss sich innerhalb der Klasse der Liste befinden.

Komplexes JPL-Muster, das die oben beschriebenen Anforderungen abbildet:

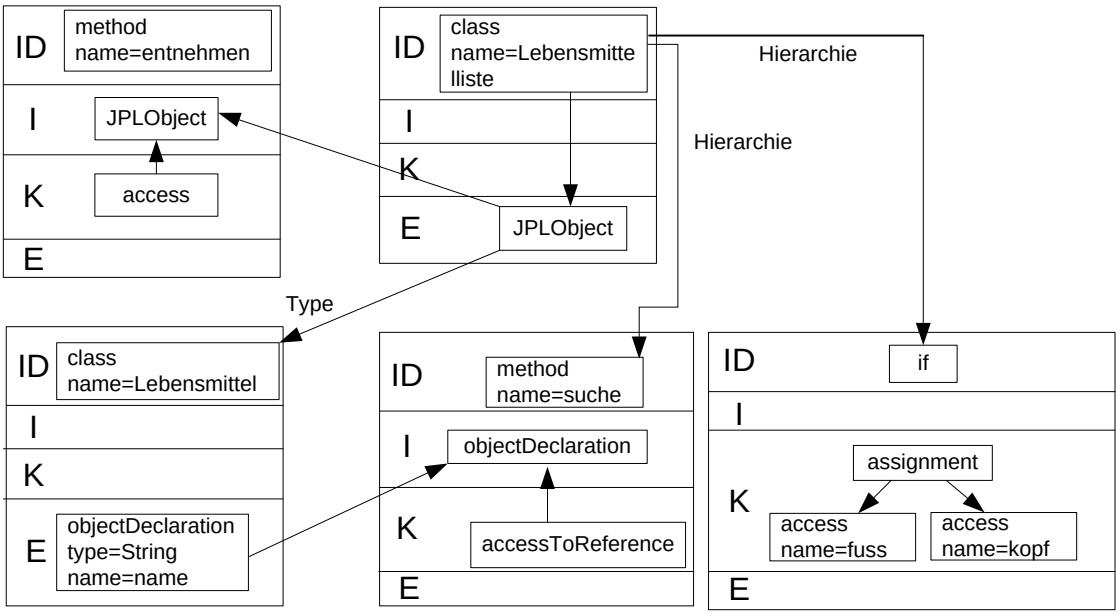


Abbildung 84: Lebensmittellager, JPL-Muster 3

In Abbildung 84 wird ein komplexes Suchmuster dargestellt, welches verschiedene Gültigkeitsbereiche umfasst. Durch dieses Muster werden die oben beschriebenen Anforderungen abgedeckt. Kennzeichnend für die Anforderungen ist, dass diese direkt oder indirekt auf das Wurzelelement, bzw. den Typ des Wurzelelements der Lagerliste referenzieren.

Im Modul, welches den Gültigkeitsbereich der Klasse der Liste umfasst, wird über die Spezifikation eines *JPLObject*-Knotens im Export des Moduls eine Membervariable gesucht, deren Referenz in die verschiedenen Bereiche übergeben wird. Da die Werteübergabe in den Gültigkeitsbereich über einen JPL-Schnittstellknoten erfolgt, werden hier z.B. die Varianten „direkte Ansprache einer Membervariablen“ oder „Übergabe mittels Getter- oder Setter-Methode“ erfasst.

In die Klasse der Liste eingebettet ist die Methode *suche*, innerhalb deren Gültigkeitsbereich auf die Variable *name* des zu verwaltenden Elementtyps referenziert wird. Ziel ist hier zu analysieren, ob die Variable *name* der Klasse verwendet wird, von deren Typ auch das Wurzelelement ist. Falls dies nicht der Fall ist, war die Implementierung der Methode *suche* wahrscheinlich falsch. Weiterhin wird nach einer *if*-Bedingung gesucht, welche in die Klasse, die die Methode *suchen* beinhaltet, eingebettet ist und in deren Gültigkeitsbereich eine direkte Zuweisung der Membervariablen *kopf* und *fuss* vorgenommen wird. Der Typ *access* wird als Abkürzung für *accessToReferenceTypeVariable* verwendet. Hierdurch wird zum einen

erkannt, dass die Struktur an sich verwendet wurde (wie in der Aufgabenstellung für die Erstzuweisung eines Elements an die Liste gefordert) und dass diese in der Listenklasse implementiert wurde, so dass die Wahrscheinlichkeit hoch ist, dass sie im Rahmen der *einfügen*-Methode genutzt wurde. Abschließend wird untersucht, ob die Methode *entnehmen*, welche in keiner hierarchischen Beziehung zur Klasse der Liste stehen muss, dass Wurzelelement der Liste korrekt importiert, und somit der Entnahmevorgang auf das richtige Element verweist. Der Typ des Zugriffs wird durch Nutzung des JPL-Schnittstellenknotens ermittelt und zurückgegeben. Somit kann bestimmt werden, ob die Methode innerhalb oder außerhalb der Lagerverwaltung implementiert wurde. Da die Übergabe abstrakt definiert wurde, ist ein Zugriff auf das Wurzelelement sowohl unter direkter Ansprache einer Membervariablen als auch die Nutzung einer Getter-Methode als akzeptierte Implementierungsvariante möglich.

Die Anwendung des Musters erzeugt u.a. folgende Analyseergebnisse, welche eine nachfolgende manuelle Prüfung unterstützen oder als Parameter genutzt werden können, um zu entscheiden, welche Muster nachfolgend gesucht werden sollen, um weitere Aspekte der Lösung zu testen.

- Wurde eine der Implementierungsvarianten in der Lösung gefunden? Falls dies nicht der Fall ist, so ist die Lösung wahrscheinlich fehlerhaft.
- Welcher Übergabetyp wurde für den Zugriff auf die Wurzel implementiert? Hier könnte ein Übungsleiter z.B. die Verwendung einer Übergabe mittels Getter-Methode in der Methode *entnehmen* als qualitativ hochwertiger bewerten als die direkte Ansprache einer Membervariablen, auch wenn beide als korrekt erkannt werden. So werden auch Aussagen über die Codequalität unterstützt.

Vergleich JPL-Muster zu bestehenden AST-Mustern:

Während die bestehenden Regeln zu der Übung eher lokale Strukturen, wie die Nutzung von Schleifen und den Durchlauf der Liste über ein *next*-Element, erfassen, wird durch dieses JPL Muster der Fokus auf die Beziehungen zwischen den Bereichen gelegt, bzw. auf die Verwendung der korrekten Objekte, welche außerhalb der Gültigkeitsbereiche der Methoden liegen (Wurzelobjekt und Name des eingelagerten Warentyps).

Dieses, im Vergleich zur Suche nach „einfachen lokalen“ AST-Mustern, aufwendigere Verfahren liefert Ergebnisse über mögliche Probleme bei der Werteübergabe zwischen Gültigkeitsbereichen, die eine umfangreichere Bewertung der Lösung über Methodengrenzen hinweg ermöglichen.

Wie bereits im Beispiel zur *mathematischen Funktion* hat auch hier der Student, im Vergleich zu ursprünglichen Aufgabe, einen höheren Freiheitsgrad bzgl. der möglichen richtigen Lösungen erhalten.

8.3. Geometrische Objekte

Aufgabenstellung:

Entwickeln Sie die Klassen *Quadrat* und *Pyramide*, welche die beiden Methoden *berechneFlaeche* (berechnet die gesamte Fläche eines Objektes) und *berechneKantenlaenge* (bildet die Summe über alle Kantenlängen des Objekts) implementieren. Verwenden Sie im Konstruktor genau einen Parameter für Kantenlänge bei Quadrat und genau zwei Parameter für Kantenlänge und Höhe bei der Pyramide.

Entwickeln Sie in diesem Kontext eine Klasse *Wuerfel*, welche die Methoden für dieses geometrische Objekt erweitert.

Analyse der funktionalen Anforderungen:

Die Lösungsstruktur kann ausgehend von den geforderten Klassen und Methoden in neun Module unterteilt werden: die Klassen *Würfel*, *Quadrat*, *Pyramide*, sowie pro Klasse der Konstruktor und die Methoden *berechneFlaeche* und *berechneKantenlaenge.*

Betrachtete Anforderung mit Fokus auf die Vererbungsbeziehungen:

In der Methode *berechneFlaeche* des Würfels muss zur Berechnung eine Variable verwendet werden, welche die Kantenlänge enthält und die im Konstruktor belegt wird.

Hinweis: Die Aufgabenstellung gibt keine Struktur für die Implementierung der Variablendeklaration vor, während in der ursprünglichen Aufgabe explizit eine Vererbungsstruktur gefordert war. Diese Lösung wird im gegebenen Kontext zwar bevorzugt, ist aber nicht die einzige zulässige Implementierungsvariante. Dies bedeutet, dass die ursprüngliche Aufgabenstellung dahingehend erweitert wurde, dass der Student selbst entscheiden kann, welches die angemessene Lösungsstruktur für die Aufgabe ist. Er muss also erkennen, dass in diesem Kontext die Vererbungsstruktur gewählt werden sollte.

Zur Analyse, ob diese Anforderung im vorgegebenen Kontext korrekt implementiert wurde, ist es notwendig zu ermitteln, in welchem Gültigkeitsbereich die Kantenlänge im Lösungscode deklariert wurde. Möglich sind hier z.B. die Übergabe als Methodenparameter, Membervariable der Klasse *Würfel* oder als Membervariable der Klasse *Quadrat*, welche vererbt wurde, oder als Membervariable der Klasse *Quadrat*, welche aber in keiner Vererbungsbeziehung zur Klasse *Würfel* steht.

Muster zur Prüfung, ob die Anforderung umgesetzt wurde:

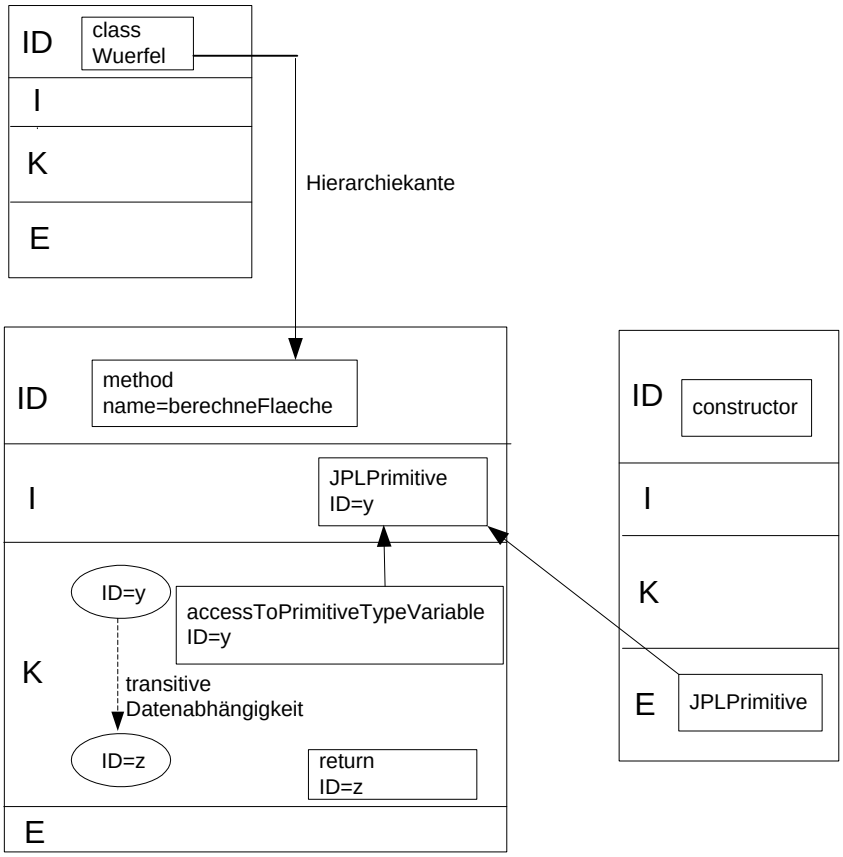


Abbildung 85: Geometrisches Objekt, JPL-Muster 4.

Das Muster erfasst folgende Aspekte:

1. In der Methode *berechneFlaeche* wird eine Variable verwendet, die in die Methode importiert wird und Einfluss auf das Ergebnis der Methode hat.
2. Durch die Hierarchiekante wird sichergestellt, dass die richtige Methode identifiziert wird. In der Vererbungsstruktur existieren mehrere Methoden mit dem gleichen Namen.
3. Weiterhin wird die Variable aus dem Konstruktor heraus exportiert, d.h. mit einem Wert belegt, der nachfolgend in der Methode ausgelesen werden kann. Da die Analyse statisch erfolgt und in dieser Aufgabenstellung der direkte Zugriff auf eine Membervariable (die Kantenlänge) betrachtet wird, kann die zuvor dargestellte Aufrufabfolge nicht überprüft werden, sondern lediglich deren strukturelle Möglichkeit (Belegung im Konstruktor und Nutzung in der Methode *berechneFlaeche*).
4. Der Typ der Variablenübergabe und der Gültigkeitsbereich, in dem diese Variable deklariert wird, sind offen. Wird eine Implementierungsvariante gefunden, so wird diese als Ergebnisinformation ausgegeben.

Alternative korrekte Implementierungsvarianten

Variante 1: Vererbungsstruktur mit Zugriff auf die vererbte Kantenlänge

```
class Quadrat{
int Kantenlaenge;
public Quadrat(int k){Kantenlaenge=k;}
…}

class Wuerfel extends Quadrat{
berechneFlaeche{
    …=Kantenlaenge…;
}…
```

Variante 2: Die Kantenlänge wird in der Klasse Würfel deklariert

```
class Wuerfel {
int kantenlaenge

public Wuerfel(int k){kantenlaenge=k;}
berechneFlaeche{
    …=Kantenlaenge…;
}…
```

Variante 3: Die Kantenlänge wird in der Klasse Quadrat deklariert. Für die Verwaltung wird ein Objekt erstellt. Eine Vererbungsstruktur wird nicht implementiert.

```
class Quadrat{
int Kantenlaenge;
public Quadrat(int k){Kantenlaenge=k;}
…}

class Wuerfel {
Quadrat q = new Quadrat(12); //12 ist die im Konstruktor übergebene Kantenlänge
berechneFlaeche{
    …=q.Kantenlaenge…;
}…
```

Abhängig von der gefundenen Implementierungsvariante können weitere manuelle Analysen ausgeführt werden oder dieses Ergebnis kann als Argument für eine weitere automatisierte Prüfung dienen. Zum Beispiel kann bei einer identifizierten Vererbungsstruktur, d.h. die Klasse *Würfel* erbt die Variable von der Klasse *Quadrat*, im Weiteren geprüft werden, ob diese Variable im Konstruktor unter Verwendung der Anweisung *super()* belegt wurde.

Vergleich JPL zu bestehenden AST Mustern:

Die bestehenden AST Muster zielen auf die korrekte Erstellung der Vererbungsstruktur ab. Da die Aufgabe in diesem Kapitel so modifiziert wurde, dass auch weitere Strukturen möglich sind, kann ein direkter Vergleich hier nicht gezogen werden. Während die bestehenden AST- basierten Regeln überprüfen, welche Strukturen nicht auftreten dürfen, wird über das JPL-Muster zusätzlich analysiert, welche Codestruktur zur Variablenübergabe verwendet wurde, um durch diese Information die qualitative Bewertung der Lösung zu unterstützen. Eine Lösung, welche die Vererbungsstruktur verwendet, könnte z.B. als qualitativ besser bewertet werden als eine Lösung, welche alle Attribute, die bereits im Quadrat deklariert wurden, nochmals im Würfel deklariert.

8.4. Verwaltung geometrischer Objekte als Liste

Aufgabenstellung:

In diesem Zusatzprojekt für Programmierbegeisterte werden die Inhalte aller vorherigen Projekte in einer Aufgabe zusammengefasst.

Gegeben sei eine abstrakte Klasse *GeometrischesObjekt*, die die abstrakte Methode *berechneKantenlaenge* (berechnet die gesamte Kantenlänge eines Objektes) definiert.

1. Entwickeln Sie zunächst die Klasse Quadrat, die diese Klasse erweitert. Verwenden Sie eine Membervariable für die Kantenlänge.

2. Entwickeln Sie anschließend eine Klasse Wuerfel, die das Quadrat erweitert.

3. Implementieren Sie weiterhin die Klasse Container, welche geometrische Elemente verwalten kann. Ein Nutzer soll die gewünschten Elemente zunächst erzeugen und dann in den Container einfügen. Der Container soll eine Methode enthalten, die die Gesamtfläche aller erfassten Objekte angibt. Ein Container kann weitere Container enthalten.

Nutzen Sie für die Realisierung folgende Vorlage:

```
public abstract class GeometrischesObjekt {
public abstract double berechneKantenlaenge();
}
public class Container {
GeometrischesObjekt g;
}
```

Anforderungsanalyse:

Da das Ziel dieses Kapitels die gemeinsame Betrachtung der Verwaltung der Objektliste und der einzelnen geometrischen Objekte ist, wird im Folgenden ein Muster dargestellt, welches Anforderungen über diese Bereiche umfasst.

1. Die geometrischen Objekte werden in einer Membervariablen der Klasse Container gespeichert.
2. Zur Berechnung der Gesamtkantenlänge ist eine Schleife über diese Objekte notwendig.
3. Zur Berechnung muss von jedem geometrischen Objekt in dieser Schleife dessen Kantenlänge berechnet und übergeben werden. Dies wird am Beispiel des Würfels untersucht.
4. Zur Berechnung der Summe der Länge aller Kanten muss die Kantenlänge (Länge einer einzelnen Kante) verwendet werden.

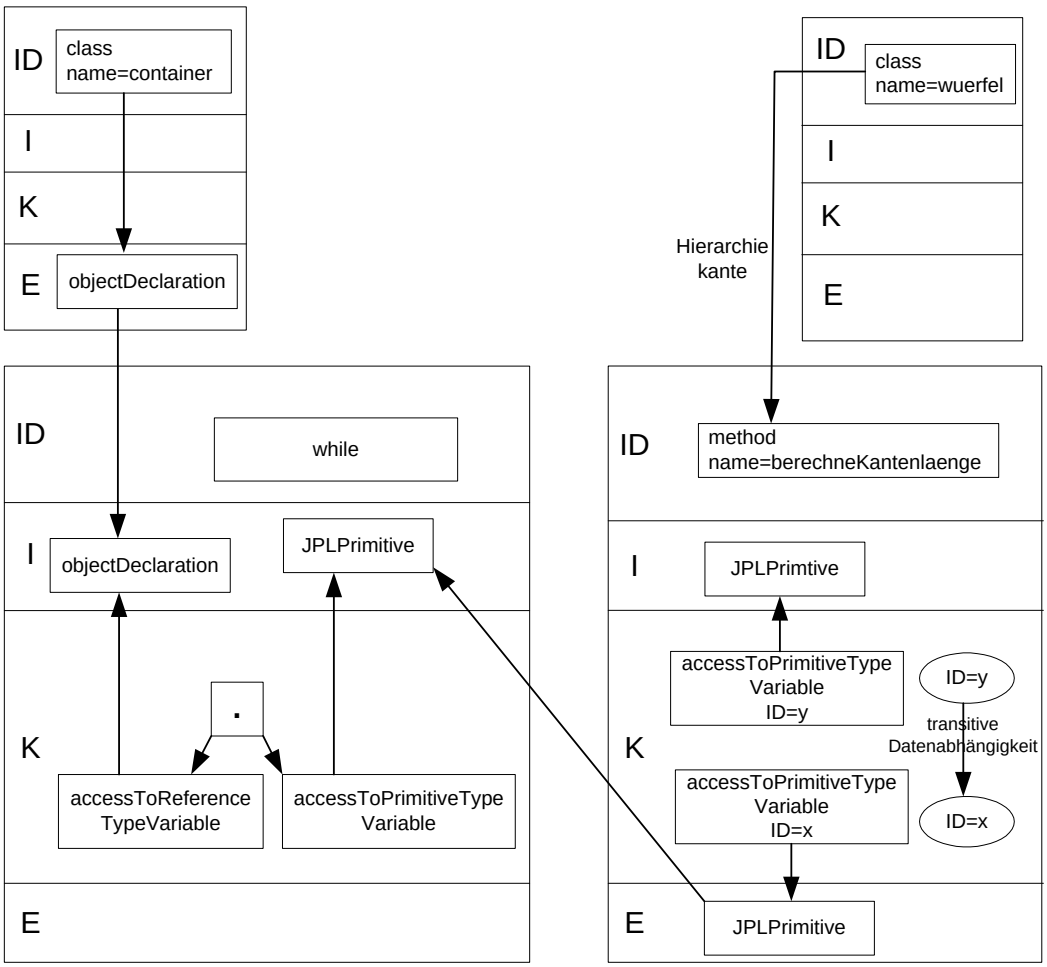


Abbildung 86: Geometrische Objekte über Liste, JPL-Muster 5.

Dieses Muster beschreibt vier Module. Über das erste Modul links oben wird sichergestellt, dass im Gültigkeitsbereich des zweiten Moduls links unten, einer *while*-Schleife, eine Membervariable der Klasse *Container* verwendet wird (hier ist das, bzw. je nach Implementierung, sind die geometrischen Objekte gespeichert). Aus diesem Objekt wird ein Wert übergeben, wobeit die Art der Übergabe offen ist. Der Wert kann z.B. von der Methode *berechneKantenlaenge* über eine *return*-Anweisung, oder durch eine globale Variable der Klasse *Wuerfel* übertragen werden.

Die Berechnung der Gesamtkantenlänge in der Klasse *Würfel* erfolgt über eine importierte Variable (der Kantenlänge). Im Ergebnis wird ausgegeben, woher die Kantenlänge (Länge einer einzelnen Kante) importiert wird. Möglich ist hier z.B. ein Import über eine Membervariable der Klasse *Quadrat* oder der Import über eine als „Getter"-Methode identifizierte Struktur der Klasse *Quadrat* (Vererbungsstruktur).

Nach Anwendung des Musters ergeben sich zwei Ergebnisse, welche weitere Analysen unterstützen:

1. Auf welche Art die Gesamtkantenlänge des Objekts an die *while* Schleife in der Methode, welche die Gesamtkantenlänge aller Objekte des Containers berechnet, übergeben wurde (über einen Return-Parameter oder eine globale Variable).

2. Auf welche Art die Kantenlänge an die Methode zur Berechnung der Gesamtkantenlänge des Objekts übergeben wurde (Membervariable der Klasse *Wuerfel*, Membervariable der Klasse *Quadrat*, über eine Methode der Klasse *Quadrat*, etc.).

Die Art der Übergabe kann als Bewertungskriterium zur Qualität der Lösung genutzt werden. So ist z.B. die Implementierung einer Vererbungsstruktur evtl. höher einzustufen als die Nutzung einer lokalen Variablen in der Klasse *Wuerfel*. Weiterhin kann diese als Kriterium zur manuellen Analyse genutzt werden. So ist eine direkte Übergabe der Objektfläche als *return*-Parameter evtl. höher einzustufen als die Übergabe einer globalen Variablen.

Für die hier beschriebene Übung wurden in der Vergangenheit nur sehr allgemeine AST-Muster erstellt. Ein Grund lag in der großen Anzahl von Implementierungsvarianten, welche für die Lösung dieser Aufgabe möglich sind. Das in diesem Kapitel dargestellte Muster zeigt, wie durch die JPL ohne zu großen manuellen Aufwand ein Muster erstellt werden kann, das zur Bewertung der Lösung über einen größeren strukturellen Kontext beitragen kann. Dies ist nur möglich, da durch die JPLPrimitive-Struktur eine Vielzahl an Implementierungsvarianten gekapselt wird. Weitere JPL-Muster für eine genauere Bewertung geforderter Vererbungsstrukturen oder zur Analyse weiterer Methoden, sind denkbar. Diese könnten z.B. an die Strukturen der Suchmuster in Kapitel 8.2. und 8.3. angelehnt werden.

8.5. Kriterien zu geeigneten Übungsaufgaben und Hinweise zur Erstellung von Suchmustern

In diesem Kapitel werden Kriterien aufgeführt anhand derer eingeschätzt werden kann, welche Art von Übungsaufgaben mittels der JPL gut überprüft werden können, bzw. in welchen Bereichen die JPL nur bedingt einsetzbar ist. Weiterhin werden Hinweise für die Erstellung von JPL-Strukturen gegeben. Die Erkenntnisse resultieren aus der Konzeption und Implementierung der JPL-Muster zu den oben dargestellten Übungsaufgaben.

Kriterien für geeignete Aufgabentypen

- Strukturbezogene Ausrichtung: Der Fokus der Aufgabe sollte auf die Verwendung spezieller Strukturen ausgerichtet sein. Hier können z.B. Aufgaben, an denen Vererbungsbeziehungen geübt werden sollen, die Verwendung von bestimmten Bedingungsstrukturen oder Methoden, bei denen jede eine festgelegte Anforderung realisiert und strukturellen Bezug zu weiteren Methoden und Klassen hat (z.B. eine Aufgabe, die ein Framework nutzt, dessen Methoden in Bezug zueinander stehen und entsprechend in einer vorgegebenen Struktur aufgerufen werden müssen) genannt werden.
- Übergreifende Anforderungen: Die Übungsaufgabe sollte bereichsübergreifende Anforderungen enthalten, d.h. eine Anforderung wird über mehrere Gültigkeitsbereiche hinweg realisiert, die in engem Bezug zueinander stehen. Der Bezug wird durch Werteübergaben realisiert.

- Nicht zu komplex und nicht zu klein:
    - Anforderungen, die in einem begrenzten Rahmen, wie z.B. einer Methode, gelöst werden können, liegen nicht im Fokus der JPL. Durch die Verbindung von JCG- und AJSDG Elementen in der Körpersektion der Module werden auch „lokale“ Muster zur Suche nach Strukturen in diesem Bereich unterstützt. Allerdings ist besonders der modulbasierte Aufbau der Sprache mit den Sektionen zum Import/Export und der Gültigkeitsbereichsidentifikation auf die Spezifikation von bereichsübergreifenden Suchmustern ausgerichtet und bildet in diesem Kontext einen Overhead.
    - Aufgaben, die sich nur auf den strukturellen Aufbau im Großen, wie z.B. die Struktur der Klassen und Methodenköpfe richten, können über den modularen Aufbau der JPL erfasst werden. Allerdings wird hier nur die Identifikatorsektion der Module genutzt, so dass ein großer Overhead bei Nutzung der modulbasierten JPL entsteht.

  Anforderungen, deren Lösungsstrukturen über die JPL formuliert und nachfolgend gesucht werden, sollten eine Verbindung von Strukturen im Kleinen und Strukturen im Großen beinhalten, die gemeinsam gesucht werden.
- Offene Variablenübergaben: Die Aufgaben sollten möglichst offen lassen, auf welche Art die Werteübergabe zwischen den verschiedenen Quelltextteilen, die die Anforderungen realisieren, implementiert wird. Diese Übergabestrukturen können über die JPL-Schnittstellenstrukturen nachfolgend identifiziert und vom Übungsleiter bewertet werden.
- Dynamische Strukturen: Übungsaufgaben sind nicht geeignet zur Prüfung durch die JPL, bzw. generell für die statische Analyse, wenn diese einen großen Anteil an dynamischen Strukturen besitzen. Zu nennen ist hier z.B. dynamisches casten (Festlegen des Objekttyps) von Objekten zur Laufzeit.

Hinweise zur Erstellung geeigneter Suchmuster:

Wie schon zuvor in der Arbeit beschrieben, liegt die Spezifikation von Suchmustern im Spannungsfeld zwischen einer zu abstrakten Spezifikation mit der Gefahr von *false positives* und einer zu konkreten Spezifikation mit der Gefahr von *false negatives* und einem hohen Spezifikationsaufwand. Im Folgenden werden darüber hinaus einige Hinweise zur Erstellung von Suchmustern zusammengestellt, die aus den Erfahrungen im Rahmen der Konzeption und Implementierung der zuvor dargestellten Muster resultieren:

- Detaillierte Identifikatorsektion oder detaillierte Körpersektion: Um die Anzahl der Matches möglichst gering zu halten, sollte versucht werden, entweder die Identifikatorsektion oder den Graph in der Körpersektion möglichst detailliert zu spezifizieren. So kann entweder ein allgemeines Muster in einem stark abgegrenztem Bereich oder ein sehr genau spezifiziertes Muster in einem umfangreichen Bereich gesucht werden.
- Werteübergabe über JPL-Schnittstellenknoten spezifizieren: Da ein Fokus der JPL auf der automatischen Identifikation der verwendeten Übergabestrukturen liegt, wodurch der Spezifikationsaufwand erheblich sinkt, sollte diese Struktur

zur Beschreibung von Variablenübergaben eingesetzt werden. Nur bei fester Vorgabe der Übergabe in der Aufgabenstellung sollten direkt die entsprechenden JCG-Strukturen genutzt werden.

- Konzeptionell muss vor der Spezifikation des Suchmusters analysiert werden, ob es für die Prüfung der betrachteten strukturellen Anforderung notwendig ist, eine Struktur über mehrere Bereiche aufzuspannen, die im Ganzen erfasst werden sollen. Evtl. ist es zur Analyse ausreichend voneinander unabhängige Teilmuster zu erfassen.
- Wenn die Identifikatorsektion nur allgemein beschrieben werden kann (z.B. nur als *while*-Schleife), sollte versucht werden eine hierarchische Beziehung zu einem weiteren Modul zu spezifizieren, um den Suchraum einzugrenzen.
- Knoten in Zusammenhang mit anderen Elementen setzen: Um die Anzahl der möglichen Matches zu begrenzen, sollten in der Körpersektion keine isolierten Knoten spezifiziert werden, sondern diese möglichst in Zusammenhang mit weiteren Elementen gebracht werden.
- Doppelte Strukturen vermeiden: Wenn in einem Muster sowohl Elemente des JCG- als auch des AJSDG verwendet werden, muss darauf geachtet werden, dass eine Beziehung nicht doppelt spezifiziert wird; z.B. ein Pfad zwischen zwei JCG- Knoten, zwischen deren AJSDG-Anweisungen bereits eine transitive Kontrollabhängigkeitskante besteht.
- Falls AJSDG-Knoten durch JCG-Knoten konkretisiert werden sollen, so muss abgewogen werden zwischen der steigenden Wahrscheinlichkeit, dass kein *false positive* gefunden wird, und dem Verlust von Implementierungsvarianten, die ohne die Konkretisierung erkannt würden. Beispiel: In Übungsbeispiel 1 „Berechnung einer Funktion“ wird ein AJSDG-Knoten durch ein JCG-Schleifenelement konkretisiert. Hierdurch wird die Rekursion als mögliche Lösung nicht mehr gefunden, welche ohne die Einschränkung auf Schleifenstrukturen als gültiges Muster akzeptiert worden wäre. Die Prüfung, ob und auf welche Art die Funktionsergebnisse aufsummiert werden, wäre ohne diesen JCG-Knoten nicht mehr Teil des Musters gewesen.
- Kleinstmöglichen Gültigkeitsbereich in der Identifikatorsektion angeben: Da alle Knoten in den erkannten Gültigkeitsbereichen markiert werden müssen, sollte immer der kleinstmögliche bekannte Gültigkeitsbereich gewählt werden, um den Markierungsvorgang möglichst kurz zu halten. Hierbei ist abzuwägen, ob die Identifikation evtl. mehrfach vorkommender kleinerer Gültigkeitsbereiche performanter ist als die einmalige Betrachtung und Markierung eines größeren Bereichs.

Dieses Kapitel bildet den Abschluss der Auswertung der praktisch realisierten Anwendungsbeispiele. Es wurde die Realisierbarkeit des Prozesses von der Spezifikation des JPL-Musters bis zur Erstellung der Ergebnisknoten gezeigt, außerdem wurde die im Rahmen der praktischen Mustererstellung gewonnen Erkenntnisse zu Kriterien für Übungsaufgaben und Hinweise zur Musterspezifikation aufgeführt.

8.6. Zusammenfassung

Der Fokus dieses Kapitels lag in der Darstellung typischer Muster, welche im Rahmen von Übungsaufgaben unter Einsatz der JPL erstellt werden können. Hierzu wurden bestehende Java-Übungsaufgaben so abgeändert, dass die signifikanten Aspekte, welche JPL-Muster von Mustern, die den reinen AST verwenden, abheben, beschrieben werden konnten. Alle gezeigten Muster wurden über das AGG Tool implementiert und die Mustersuche auf den unterschiedlichen Übungslösungen automatisiert ausgeführt. Damit wurde die praktische Anwendbarkeit des Ansatzes belegt.

Vorteile der JPL-Muster, welche durch die Beispiele dieses Kapitels gezeigt wurden:

- Beschreibung der Muster basierend auf einer modularen Darstellung, so dass eine modul- und sektionsorienterte Strukturierung der Suchmuster möglich wird und die Gesamtmuster aus einzelnen separaten Modulen strukturiert zusammengesetzt werden können. Somit wird eine schrittweise Erfassung des Gesamtmusters unterstützt.
- Spezifikation des Suchbereichs in der Identifikatorsektion. Bei einer Spezifikation direkt über den AST ist bereits die Markierung der zu untersuchenden Bereiche mit erheblichem Aufwand verbunden.
- Nutzung von Daten- und Kontrollabhängigkeiten, wo eine reine AST-Beschreibung aufgrund der hohen Anzahl möglicher Implementierungsvarianten nicht möglich ist. Zur Konkretisierung der Muster können diese nachfolgend um JCG-Elemente ergänzt werden.
- Spezifikation von abstrakten JPL-Schnittstellenstrukturen, um mehrere Implementierungsvarianten zur Werteübergabe zwischen Gültigkeitsbereichen zu kapseln, so dass nicht eine Vielzahl an AST-Mustervarianten spezifiziert werden muss. Hierdurch können den Studenten größere Freiheitsgrade bzgl. der Lösung gewährt werden. Weiterhin wird als Analyseergebnis die Art der Übergabe bei einer korrekten Lösung ausgegeben. Diese Information kann entweder als Grundlage für weitere Analyseschritte oder als Basis für eine Bewertung der Qualität der Lösung dienen.

Abschließend wurden auf Grund der gesammelten Erkenntnisse in der Konzeption und Implementierung der Suchmuster Kriterien für geeignete Übungsaufgaben und Hinweise zur Erstellung von Suchmusterstrukturen gegeben, so dass ein Nutzer den Anwendungsbereich der JPL besser einschätzen und seine Muster entsprechend spezifizieren kann.

9. Fazit

Das Kapitel beschreibt zusammenfassend den Beitrag, den diese Arbeit in der Domäne der statischen Mustererkennung auf Java-Quellcode leistet. Nachfolgend wird die Arbeit einer kritischen Analyse unterzogen und abschließend ein Ausblick auf Erweiterungsmöglichkeiten des hier vorgestellten Ansatzes gegeben.

9.1. Beitrag der Arbeit

Das Ziel dieser Arbeit war die Entwicklung eines graphbasierten Ansatzes, der die Erstellung und die nachfolgende Anwendung komplexer Suchmuster, über mehrere Gültigkeitsbereiche eines gegebenen Java-Quelltextes hinweg, unterstützt. Der engere Fokus lag auf der Mustersuche in Java-Übungsaufgaben, wie sie typischerweise in Lehrveranstaltungen eingesetzt werden. Besondere Herausforderungen stellten in diesem Zusammenhang die Behandlung von Implementierungsvarianten und die strukturierte Erstellung eines modularen Suchmusteraufbaus dar.

Um dieses Ziel zu erreichen, wurde die Java Pattern Language eingeführt, die als visuelle Spezifikationssprache auf dem Abstrakten Syntax Baum und dem Systemabhängigkeitsgraphen aufbauend, eine modularisierte Musterspezifikation ermöglicht. Die JPL ist eine graphbasierte Sprache, deren Grammatik über Knoten- und Kantentypen definiert wird. Die Leistung dieser Arbeit umfasst hier die Definition der Sprachgrammatik der JPL, deren Syntax aus den Graphtypen des JCG, AJSDG, deren Erweiterungen *Pfad* und *Transitive Hülle* sowie der Definition einer abstrakten Schnittstellenstruktur besteht. Weiterhin wird eine modularisierte Musterspezifikation über diese Syntaxelemente unterstützt. Die Ableitung der abstrakten Suchmuster und die Durchführung der Mustersuche wird über die typisierte und attributierte Graphtransformation realisiert.

Folgende Konzepte wurden durch die JPL eingeführt:

1. Durch die JPL wird die kombinierte Spezifikation des Syntax nahen Java Code Graphen mit dem Daten- und Kontrollabhängigkeiten beschreibenden Adapted Java System Dependency Graphs in einem Suchmuster ermöglicht. Dies erleichtert die Erstellung komplexer Muster, da z.B. durch den parallelen Einsatz dieser Graphtypen Daten- und Kontrollabhängigkeiten spezifiziert werden können, welche für das komplexe Muster signifikant sind, und im Weiteren JCG-Elemente hinzugenommen werden können, um das Muster zu konkretisieren und das Risiko des Auftretens von *false positives* zu reduzieren. Ebenso können ausgehend von Mustern, die nur JCG-Elemente enthalten, Abhängigkeitsstrukturen erstellt werden, die auf Grund einer zu großen Anzahl an möglichen Implementierungsvarianten nicht über den JCG spezifiziert werden können.
2. Der modulare Aufbau der JPL ermöglicht die Erstellung eines komplexen Suchmusters ausgehend von einzelnen lokalen Mustern, welche nachfolgend miteinander kombiniert werden. Somit ist es nicht notwendig, während der

Spezifikation immer das vollständige Muster betrachten zu müssen. Auch die arbeitsteilige Erstellung eines Suchmusters wird somit möglich.

3. Die Identifikator-Sektion der JPL-Module ermöglicht es den Suchraum einzugrenzen, in dem die im Modulkörper spezifizierte Struktur gesucht wird. Der Suchraum wird über Gültigkeitsbereiche festgelegt. So kann z.B. detailliert angegeben werden, in welcher Methode das Muster gesucht werden muss, oder nur allgemein der Gültigkeitsbereich einer Klasse spezifiziert werden.
4. Durch die Einführung der JPL-Schnittstellenstrukturen wird von der konkreten Implementierungsvariante der Variablenübergabe zwischen den durch die Muster beschriebenen Gültigkeitsbereichen abstrahiert. Hierdurch reduziert sich der manuelle Spezifikationsaufwand und die Erstellung komplexer Suchmuster wird entsprechend unterstützt.

Die Anwendung der JPL-Muster erzeugt verschiedene Ausgaben, welche die Bewertung des zu analysierenden Quellcodes unterstützen. Zusätzlich zu der Meldung, ob ein Muster gefunden wurde, wird für gefundene Musterübereinstimmungen der im Quelltext identifizierte Übergabetyp ausgegeben, falls im Muster eine abstrakte Schnittstellenstruktur angegeben wurde. Beispiele sind hier die Übergabe als Methodenparameter, der Zugriff auf eine öffentliche Membervariable, oder auf eine als *private* deklarierte Variable über Getter- und Setter-Methoden. Anhand dieser Informationen kann der Anwender entscheiden, ob eine weitere manuelle Prüfung notwendig ist.

Weiterhin wurden Vorgehensweisen zur Musterspezifikation vorgestellt, die im Rahmen der statischen Prüfung von Java Übungsaufgaben eingesetzt werden können. Die im konkreten Fall anzuwendende Vorgehensweise richtet sich nach der zur Verfügung stehenden Basis bestehender Lösungen, dem geplanten Aufwand zur Mustererstellung und dem erwarteten Prüfungsaufwand. Es wird zwischen Top-Down und Bottom-Up Ansätzen unterschieden, welche abhängig vom Aufgabenkontext verwendet werden können.

Im praktischen Teil der Arbeit wurden die Erstellung von Mustern in JPL, die Ableitung von abstrakten Schnittstellenstrukturen und die Suche nach den spezifizierten Mustern auf Grundlage des Tools AGG realisiert. Es wurden verschiedene Graphregelsätze implementiert, die die Ableitung und Mustersuche unter Nutzung von AGG unterstützen. Die Mustersuche nach JCG-Strukturen wurde bereits anhand tausender Aufgabenlösungen von studentischen Übungsaufgaben mittels des Tools JACK (Java Checker) durchgeführt, während der praktische Einsatz des AJSDG und der Ableitung der modulbasierten JPL-Musterspezifikation exemplarisch betrachtet wurde. Abschließend wurden Kriterien dargestellt, auf Grund derer entschieden werden kann, ob eine Übungsaufgabe sich zur Analyse mittels der JPL eignet und es wuden Hinweise zur Mustererstellung aufgeführt.

Im folgenden Kapitel werden verschiedene Aspekte der JPL kritisch hinterfragt und aufgezeigt, wie diese in der vorliegenden Arbeit behandelt wurden.

9.2. Kritische Analyse

Aus der exemplarischen Validierung der JPL durch deren Einsatz im Rahmen verschiedener Übungsaufgaben und dem Vergleich mit bestehenden Ansätzen zur graphbasierten Musterspezifikation ergeben sich folgende kritische Punkte:

1. Fehleridentifikation über komplexe Muster: Beim Einsatz komplexer Muster über mehrere Gültigkeitsbereiche hinweg ist die genaue Lokalisierung eines Fehlers, welcher sich auf die Struktur nur eines Bereichs bezieht, schwer durchzuführen. Über die alleinige Angabe der verwendeten Übergabetypen zwischen Bereichen können Fehler dieses Typs nicht zugeordnet werden. Zur Lokalisierung müssen die Module separat angewendet werden. In Kombination mit den Ergebnissen der Suche nach dem komplexen Muster kann nachfolgend manuell die Fehlerursache ermittelt werden. Diese Suche wird über die Anwendung der sequentiellen Ausführungsvariante, welche die Muster der einzelnen Sektionen eines komplexen JPL-Muster separat sucht, bereits implizit realisiert. Beim Einsatz der Variante, welche die Suche über Muster durchführt, in denen alle Sektionen in einem Muster zusammengefasst werden, müssen diese separaten Tests zusätzlich spezifiziert werden. Durch diese zusätzlichen Suchprozesse vermindert sich allerdings die Performance der in dieser Variante spezifizierten Mustersuche, in der die Suche nach einzelnen Teilmustern nicht vorgesehen ist. Nähere Informationen zu den Ausführungsvarianten der Mustersuche wurden in Kapitel 5.3. gegeben.
2. Performance: Die automatisierte Ableitung und Anwendung umfangreicher JPL-Muster sind sehr performanceintensiv, welches zum einen auf die Vielzahl der für die Quelltextrepräsentation zu erstellenden Graphstrukturen und zum anderen auf den Ablauf der Suche nach komplexen Mustern mit vielen Elementen zurückzuführen ist.
   Dieses Problem wurde an verschiedenen Stellen dieser Arbeit thematisiert. So wurden unterschiedliche Techniken zur Erstellung der transitiven Hülle, Ergänzungen zum JCG und alternative Abläufe zur Ausführung der Suche nach komplexen Mustern eingeführt, um dieser Herausforderung gerecht zu werden. Weiterhin wird der Suchraum durch die Angaben in der Identifikatorsektion eines Moduls eingegrenzt.

   Es können folgende grundlegende Kriterien im Rahmen der Musterspezifikation genannt werden, um diesem Problem zu begegnen:

   1. Der Suchraum muss über die Identifikatorsektion so eng wie möglich eingegrenzt werden.
   2. Strukturen in der Quelltextrepräsentation sollten nur erstellt werden, wenn diese für die Mustersuche notwendig sind.
   3. Suchmuster sollten nur Elemente enthalten, die zur Überprüfung der strukturellen Anforderungen notwendig sind.
   4. Die Typen der zu suchenden Variablenübergaben sollten soweit möglich eingeschränkt werden.
   5. Eine Analyse des zu suchenden Musters und die nachfolgende Entscheidung, nach welcher Methode das Muster abgeleitet wird (sektionsbasiert oder

vollständig), sollte nach den in Kapitel 5.3. dargestellten Kriterien durchgeführt werden.

In dieser Arbeit wurde keine allgemeingültige Lösung zur Performanceoptimierung beschrieben, so dass der Nutzer immer abhängig vom Kontext, d.h. der Gößе und dem Inhalt des Suchmusters und der zu erwartenden Größe und Art der zu untersuchenden Quelltexte entscheiden muss, wie er vorgeht.

3. Validierung: Zur weiteren Validierung des Ansatzes wären umfangreichere Tests sowohl im Rahmen der Lehre als auch im industriellen Kontext wünschenswert. Voraussetzung für diese Validierung ist die vollständige Implementierung des in Kapitel 7.2. beschriebenen Designs zur automatisierten Musterableitung und dessen Integration in die Testumgebung JACK, um somit dem Nutzer eine durchgängig automatisierte Testumgebung zur Verfügung zu stellen, in der eine größere Anzahl an Mustern spezifiziert und angewendet werden kann.

9.3. Ausblick

Im Rahmen der Weiterentwicklung des in dieser Arbeit präsentierten Ansatzes sollten folgende Punkte näher untersucht werden:

Da über die JPL statische Tests spezifiziert werden, ist es bei isolierter Betrachtung der Testergebnisse der Mustersuche nicht möglich, das Laufzeitverhalten der Applikationen zu analysieren. Um auch Aussagen über das dynamische Verhalten treffen zu können, ist eine Kombination dieser statischen Analyse mit dynamischen Tests notwendig, wie sie z.B. über das Tool JACK [GSB08] realisiert werden können. So könnten einem JPL-Muster eine Reihe dynamischer Tests zugeordnet werden. Erst nach erfolgreichem Durchlauf der statischen Mustersuche und der entsprechenden dynamischen Tests würde für eine Anforderung der Code als „korrekt“ bewertet und die Korrekturunterstützung erhielte eine größere Genauigkeit.

Die Erstellung des in Kapitel 6.4. vorgeschlagenen Katalogs zu typischen Suchmustern, welche im Bereich der Übungsaufgaben einzusetzen sind, könnte im Bereich der Lehre weiterverfolgt werden. Hierbei sollten sowohl die in dieser Arbeit im Fokus stehenden notwendigen Strukturen für eine korrekte Lösung als evtl. auch typische fehlerhafte Strukturen untersucht werden.

Während in der vorliegenden Arbeit der Anwendungsfokus der JPL auf der Analyse von Übungsaufgaben liegt, kann in weiteren Arbeiten die Übertragung dieses Ansatzes auf Domänen beschrieben werden, in denen die Suche nach Strukturen im Quellcode über größere Bereiche hinweg notwendig ist. Beispiele sind hier die Suche nach sicherheitskritischen Strukturen im Quellcode, die Analyse von Quelltext zum Design Recovery, oder die Analyse der Nutzung von Frameworks. Evtl. kann eine Kombination mit weiteren Analysemethoden die Geschwindigkeit der Ausführung der Mustersuche auf größeren Quelltexten weiter erhöhen. So könnten z.B. über die JPL Bereiche identifiziert werden, in denen nachfolgend über reguläre Ausdrücke weitere Teilmuster gesucht werden.

Der in dieser Arbeit vorgestellte Ansatz unterscheidet im Ergebnis der Suche lediglich zwischen einem gefundenen oder nicht gefundenen Suchmuster. Im Rahmen der Fehlerlokalisierung ist eine Unterteilung eines komplexen Musters in Teilmuster zu

analysieren. In [Tra07] wird ein Ansatz vorgestellt, in dem bereits die Identifikation von Teilmustern eines gegebenen Suchmusters für einen gültigen Match ausreicht. Nachfolgend wird anhand von Gewichtungen der gefundenen Teilmuster die Güte des Matches bewertet. Eine Kombination dieses Ansatzes mit der JPL könnte eine Methode darstellen, um die JPL auch in Domänen einzusetzen, in denen es ausreichend sein kann Teile eines komplexen Muster zu identifizieren, um wichtige Informationen über gegebenen Code zu gewinnen. Ein Beispiel ist hier das Design Recovery zur Erhöhung des Verständnisses des Quellcodes.

Über die JPL wird die Spezifikation komplexer Muster unterstützt. Die Güte der hiermit spezifizierten Muster ist jedoch abhängig vom Anwender. Während über die Nutzung der Technik der Graphtransformation implizit eine Konsistenzprüfung vorgenommen wird (z.B. dürfen keine *hängenden Kanten* spezifiziert werden), sind hier weitere eher semantikorientierte Spezifikationsprüfungen denkbar, wie z.B. die Suche nach Datenflussanomalien (z.B. Spezifikation zweier Variablenbelegungen, welche direkt aufeinander folgen) in JPL-Spezifikationen.

Die Syntax der JPL ist an der Programmiersprache Java ausgerichtet, da auch detaillierte Programmkonstrukte über ein Suchmuster erfasst werden sollen und somit entsprechend spezifiziert werden müssen. In zukünftigen Arbeiten kann dieser Ansatz auf weitere Programmiersprachen erweitert werden. In der Aufwandsbetrachtung zu dieser Anpassung muss zwischen dem konzeptionellem Ansatz und der praktischen Realisierung unterschieden werden. Die Konzeption der zu spezifizierenden Graphen ist bereits so ausgelegt, dass eine Anpassung ohne zu großen Aufwand möglich sein sollte. So ist der SDG sprachenunabhängig und die Strukturen und Befehle des JCG, wie z.B. Methoden und Schleifenkonstrukte, sind so auch in anderen Programmiersprachen zu finden, und können hier wiederverwendet werden. Diese müssen um fehlende Konstrukte, wie z.B. Pointer ergänzt werden. Die praktische Realisierung dieser Erweiterung erfordert für die Erstellung des ASTs Parser der entsprechenden Sprache, über welche die angepassten Graphstrukturen zur Quellcoderepräsentation generiert werden können. Der nachfolgende Aufbau des AJSDG, die Ableitung der Suchmuster und die Durchführung der Suche ändern sich abhängig von den Variablenübergabestrukturen der zu betrachtenden Sprache. So können z.B. bei einer Übertragung auf die Sprache C++ viele Strukturen wiederverwendet werden, da es sich hier ebenfalls um eine objektorientierte Sprache handelt.

Um die JPL im größeren Rahmen einsetzen zu können, ist es notwendig, sowohl den in dieser Arbeit beschriebenen JPL-Editor, als auch die automatisierte Mustersuche über JPL- Muster in einer Systemumgebung zu implementieren. Diese Testspezifikations- und Testausführungsumgebung könnte nachfolgend in ein Entwicklungstool integriert werden, um somit bereits während der Implementierung einer Applikation statische Tests zu komplexeren Mustern spezifizieren und durchführen zu können. Während die für diese Arbeit erfolgte Implementierung auf dem Tool AGG basiert, sollte für eine Fortsetzung der Entwicklungstätigkeit die Nutzung weiterer Graphtransformationssprachen untersucht werden. Ein interessantes Projekt ist hier Henshin [Hen14], welches einen Graphtransformationsansatz realisert der auf dem Eclipse Modeling Framework basiert und als Eclipse Projekt eine breite Unterstützung besitzt.

10. Literaturverzeichnis

[agg14] Die AGG Homepage, http://user.cs.tu-berlin.de/~gragra/agg/, zuletzt besucht: 4.1.2014

[AFC98] Antoniol, Giuliano; Fiutem, Roberto; Cristoforetti, Lucas: Design Pattern Recovery in Object-Oriented Software. In: Proc. Of the 6th International Workshop on Program Comprehension (IWPC), Ischia, Italy, IEEE Computer Society Press, Juni 1998, S. 153–160

[BBC08] Ballis, D.; Baruzzo, A.; Comini, M.: A Rule-based Method to Match Software Patterns Against UML Models. In: Electronic Notes in Theoretical Computer Science (ENTCS), Bd. 219, November, 2008

[BBS05] Blewitt, Alex; Bundy, Alan; Stark, Ian: Automatic Verification of Design Patterns in Java. In: Proc. of the 20th International Conference on Automated Software Engineering (ASE), Long Beach, USA, 2005, S. 224–232

[BEM+00] Busatto, G., Engels, G., Mehner, K. and Wagner, A.: A framework for adding packages to graph transformation systems. In Ehrig, H. et al. (editors), Theory and Application of GraphTransformation (TAGT'98), Selected Papers, Lecture Notes in Computer Science 1764, Springer-Verlag, 2000, S. 352-367

[BKK05] Busatto, Giorgio, Kreowski, Hans-Jörg, Kuske, Sabine: Abstract hierachical graph transformation. In: Mathematic Structures in Computer Science, Bd. 15, Cambridge University Press, July 2005, S. 773 – 819

[BMD+00] Balazinska, M. ; Merlo, E.; Dagenais, M.; Lague, B.; Kontogiannis, Kostas: Advanced clone-analysis to support objectoriented system refactoring. In: Proc. Working Conf. Reverse Engineering, IEEE Computer Society, 2000, S. 98-107

[BPT02] Bottoni, P.; Parisi-Presicce, F.; Taentzer, G.: Coordinated distributed diagram transformation for software evolution. In: Electronic Notes in Theoretical Computer Science, Bd. 72 (4), 2002.

[BPT04] Bottoni, P.; Parisi-Presicce, P.; Taenzer, G.: Specifying Coherent Refactoring of Software Artefacts with Distributed Graph Transformationa. In P. v. Bommel, Transformation of Knowledge, Information, and Data: Theory and Applications, Idea Group Publishing, 2004.

[BYM+98] Baxter, I. D.; Yahin, A.; Moura, L.; Sant'Anna, M.; Bier, L.: Clone detection using abstract syntax trees. In: Proc. of Int. Conf. on Software Maintenance, 1998.

[CDF+03] Cohen, E.; Datar, M..; Fujiwara, S.; Gionis, A.; Indyk, P.; Yan, X.; Han, J.: CloseGraph: mining closed frequent graph patterns. In Proc. of 9th Int. ACM SIGKDD Conf. on Knowledge Discovery and Data Mining, 2003

[CMR+97] Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic Approaches to Graph Transformation – Part I: Basic Concepts and Double Pushout Approach. In: Rozenberg, G. (ed.), Handbook of Graph Grammars and Computing by Graph Transformation, Bd. 1, Foundations World Scientific, Singapore, 1997

[DMT10] Von Detten, M; Meyer, M.; Travkin, D.: Reverse Engineering with the Reclipse Tool Suite. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010), Cape Town, South Africa, Miy 2-8, 2010, Bd. 2, New York, NY, USA, Mai 2010, S. 299 - 300

[DW02] Dudziak, T.; Wloka, J.: Tool-supported discovery and refactoring of structural weaknesses in code. Masterarbeit, Technische Universität Berlin, Februar, 2002.

[DZP09] Dong, J.; Zhao, Y.; Peng, T.: A Review of Design Pattern Mining Techniques. In: Int. Journal of Software Engineering and Knowledge Engineering (IJSEKE), 2009, Bd. 19(6), S.823–855

[EB10] Ebert, Jürgen; Bildhauer, Daniel: Reverse Engineering Using Graph Queries. In: Schürr, Andy; Lewerentz, Claus; Engels, Gregor; Schäfer, Wilhelm; Westfechtel, Bernhard: Graph Transformations and Model Driven Engineering, Springer Verlag, Bd. 5765, 2010

[ECL14] Die Eclipse Homepage: http://www.eclipse.org, zuletzt besucht: 4.1.2014

[EE96] Ehrig, H. und Engels, G.: Pragmatic and Semantic Aspects of a Module Concept for Graph Transformation Systems. In: Proceedings Graph Grammars and their Application to Computer Science, 5th Int. Workshop, Williamsburg, USA, Lecture Notes in Computer Science Bd. 1073, Springer, 1996

[EEP+06] Ehrig, H.; Ehrig, K.; Prange, U.; Taenzer, G.: Fundamentals of Algebraic Graph Transformation, EATCS Monographs in TCS, Springer, 2006.

[EHK+97] Ehrig, H., Heckel, R., Korrff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Algebraic Approaches to Graph Transformation – Part II: Single Pushout Approach and COmparison with Double Pushout Approach. In Rozenberg, G., (ed), Handbook of Graph Grammar and Computing by Graph Transformation, Vol. 1 Foundations, S. 247-312, World Scientific, Signapore, 1997

[EJ04] Van Eetvelde, Niels; Janssens, Dirk: Extending graph rewriting for refactoring. In: Proceedings of International Conference of Graph Transformation 2004. Springer Conference, September 2004.

[ERT99] Ermel, C.; Rudolf, M.; Taenzer, G.: The AGG Approach: Language and Tool Enviroment. In: Handbook of Graph Grammars and Computing by Graph Transformation, Volume II: Specification and Programming, Ed. G. Rozenberg, World Scientific, 1999, S. 551 – 603

[FMR+07] Fuss, C., Mosler, C., Rnger, U., Schutchen, E.: The Jury is still out: A Comparison of AGG, Fujaba, and PROGRES. In: Proceedings of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques, 2007

[FOW87] Ferrante, J., Ottenstein, K., Warren, J.: The program dependence graph and its use in optimization. In: Transactions on Programming Languages and Systems, ACM, Bd. 9, July, 1987, S. 319-349.

[Fow99] Fowler, M.: Refactoring: Improving the Design of Existing Programs, Addison-Wesley, 1999

[Fuj14] Fujaba Development Group, Fujaba Tool Suite. Online unter: http://www.fujaba.de, zuletzt besucht 4.1.2014

[FWM+96] Forrest Shull, Walcélio L., Melo, Victor, Basili, R.: An inductive method for discovering design patterns from object-oriented software systems. Technical Report UMIACS-TR-96-10, University of Maryland, 1996

[GEM+99] Goedicke, M., Enders, B., Meyer, T., Taenzer, G.: Tool Support for ViewPoint-oriented Software Development: Toward Integrating Multiple Perspectives by Distributed Graph Transformation. In: Proc. International Workshop and Symposium AGTIVE – Applications of Graph Transformation with Industrial Relevance, Maonastery Rolduc, Kerkrade, Niederlande, 1999

[GHJ+95] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. In: Addison-Wesley Professional Computing Series, Reading, Massachusetts, Addison Wesley Publishing Company, 1995

[GJS+05] Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, in The Java Series from the Source, 3rd edition, Addison-Wesley, 2005

[GPS+99] Grosse-Rhode, M., Parisi-Presicce, F., Simeoni, M., Taenzer, G.: Modelling Distributed Systems by Graph Transformation based on Refinement via Rule Expresions. In: Proc. International Workshop and Symposium AGTIVE – Applications of Graph Transformation with Industrial Relevance, Monastery Rolduc, Kerkrade, Niederlande, 1999

[GRe14] GreQL Homepage, http://www.uni-koblenz-landau.de/koblenz/fb4/ist/rgebert/research/Graphentechnologie/GReQL, zuletzt besucht 4.1. 2014

[GS94] Goedicke, M.; Schumann, H.: Component oriented Software Development with II. In: IST report 21/94, Frauenhofer Institute for Software-Engineering and Systems Engineering, 1994

[GSB08] Goedicke, Michael; Striewe, Michael; Balz, Moritz: Computer Aided Assessments and Programming Exercises with JACK. In: ICB Report No 28, Institute for Computer Science and Business Information Systems, University of Duisburg-Essen, 2008

[GSC91] Goedicke, M., Schumann, H; Cramer, J.: On the Specification of Software Components. In: Proc. Of the 6th Int. ACM/IEEE Workshop on Software Specification and Design, 1991

[GTS01] Goedicke, M.; Tröpfner, P.; Enders-Sucrow, B.: Hierarchical Specification of Graphical User Interfaces Using a Graph Grammar Approach. In Journal of Integrated Design & Process Science, Bd. 5(1), IOS Press Amsterdam, Niederlnade, Januar 2001

[Har07] Harman, Mark: The Current State and Future of Search Based Software Engineering. In: Proceedings of the International Conference on Software Engineering, Workshop on The Future of Software Engineering, Editor: Lionel C. Briand and Alexander L. Wolf, Minneapolis, USA, Mai 2007

[Hen14] Homepage des Henshin Projekts, http://www.eclipse.org/henshin/, zuletzt besucht: 2.4.2014

[HHT96] Habel, A., Heckel, R., Taenzer, G.: Graph Grammars with Negative Application Conditions. In: Spezial Issue of Fundamenta Informaticae, Bd. 26, 1996

[HJE+06] Hoffmann, Berthold; Janssens, Dirk; Van Eetvelde, Niels: Cloning and Expanding Graph Transformation Rules for Refactoring. In: Electronic Notes in Theoretical Computer Science, Band 152, 2006, S. 53-67

[HM10] Harman, Mark; McMinn, Phil: A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. IEEE Trans. Software Eng. Bd. 36(2), 2010, S. 226-24

[HMR+03] Hristova, M., Misra, A., Rutter, M. and Mercuri, R.: Identifying and Correcting Java Programming Errors for Introductory Computer Science Students. Proc. the 34th SIGCSE technical symposium on Computer science education, Bd. 34, ACM Press, Reno, Nevada, USA, 2003, S.153-156

[HR92] Horwitz, Susan; Reps, Thomas: The use of program dependence graphs in software engineering. In: Proceedings of the 14th international conference on Software engineering. Melbourne, Australien, 1992, S. 392-411.

[HRB90] Horwitz, Susan; Reps, Thomas; Binkley, David: Interprocedual slicing using dependence graphs. In: ACM Transaction on Programming Languages and Systems, Bd. 12(1), 1990, S. 26-60

[Jac97] Jackson, D.: A software system for grading student computer programs. In: Proc. of the twenty-eighth SIGCSE technical symposium on Computer science education, Bd. 28, ACM Press, San Jose, California, United States, 1997, S. 335-339

[JCC14] Homepage Java Compiler Compiler, http://java.net/projects/javacc, zuletzt besucht 4.1.2014

[JJA06] Whittle, John; Araujo, Joao; Moreira, Ana: Composing Aspect Models with Graph Transformations, In. Proceedings of the 2006 international workshop on early aspects at ICSE, China, 2006, ACM Press, S. 59 - 65

[KB00] Kim, H., Boldyreff, C.: A method to recover design patterns using software product metrics. In 6th International Conference, ICSR-6, Vienna, Austria, 2000.

[KG06] Köllmann, C.; Goedicke, M.: Automation of Java Code Analysis for Programming Exercises. In: Proceedings of the 3rd International Workshop on Graph based Tools at ICGT Natal, Brazil, 2006.

[KG08] Köllmann, Carsten; Goedicke, Michael: A Specification Language for Static Analysis of Student Exercises. In: Proceedings of the 23rd International Conference on Automated Software Engineering, Aquila, Italy, 2008

[KH01] Komondoor, R.; Horwitz, S.: Using slicing to identify duplication in source code. In: Proc. of 8th Int. Symp. on Static Analysis, Springer-Verlag, 2001 S. 40–56

[KK96] Kreowski, H., Kruske, S.: On the interleaving Semantics of Transformation Units – A Step into Grace. In: Proceedings Graph Grammars and their Application to Computer Science, 5th int. workshop, Williamsburg, USA, LNCS Bd. 1073, Springer 1996

[KKL+07] Köllmann, C.; Kutvonen, L.; Linington, P.F.; Solberg, A.: An Aspect-Oriented Approach to Manage QoS Dependability Dimensions in Model Driven Development. In: 3rd International Workshop on Model Driven Enterprise Information Systems at ICEIS. Funchal, Portugal, 2007

[KP84] B.W. Kernighan, R. Price, The UNIX Programming Environment. Prentice-Hall, 1984

[KP96] Kramer, Christian; Prechelt, Lutz: Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In: Proc. of the 3rd Working Conference on Reverse Engineering, Monterey, USA, 1996, S. 208–215

[Kri01] Krinke, J.: Identifying similar code with program dependence graphs. In: Proc. of 8th Working Conf. on Reverse Engineering, 2001.

[KSR+99] Keller, R. K., Schauer, R. Robitaille, S., Page, P.: Pattern-based reverse engineering of design components. In: Proc. Of the 21st International Conference On Software Engineering, IEEE Computer Society Press, 1999, S. 226-235

[LCH+06] Liu, Chao; Chen, Chen; Han, Jiawei; Yu, Philip: GPLAG: Detection of Plagiarism by Program Dependence Graph Analysis. In: Conference on Knowledge Discovery in Data Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, Philadelphia, USA, 2006, S. 872 – 881

[MA10] Maggiono, Stefano; Arcelli, Francesca: Metrics-based detection of micro patterns to improve the Assesment of Software Quality. In: Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics, New York, USA, 2010

[Mat05] Munro, Matthew James: Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. In: Proceedings of the 11th IEEE International Software Metrics Symposium, Washington, USA, 2005

[MDJ02] Mens, T.; Demeyer, S.; Janssens, D.: Formalising behaviour preserving program transformations. In Graph Transformation, Bd. 2505 of Lecture Notes in Computer Science, SpringerVerlag, S. 286--301

[Mey00] Meyer, Torsten: Dynamik Semantics Negatiation in Distributed and Evolving Software Systems: Towards Automated Semantics-Directed System Configuration, Dissertation, Universität Essen, 2000

[MT04] Mens, Tom; Tourw'e, Tom: A survey of software refactoring. IEEE Transactions on Software Engineering, Bd. 30(2), 2004, S. 126-139

[MTR06] Mens, T.; Taentzer, G.; Runge, O: Analysing refactoring dependencies using graph transformation. Software and System Modeling, 2006

[Nav01] Gonzales, Navarro: NR-grep: A Fast and Flexible Pattern Matching Tool. In: Software: Practice and Experience, Bd. 31, nr. 13, 2001, S. 1265 – 1312

[Nij07] Shi, Nija: Reverse Engineering of Design Patterns from Java Source Code, Dissertation, University of California, 2007

[NMW04] Niere, Jörg; Meyer, Matthias; Wendehals, Lothar: User-Driven Adaption in Rule- based Pattern Recognition, Forschungsbericht tr-ri-04-249, Universität Paderborn, 2004

[NSW+02] Niere, Jörg; Schäfer, Wilhelm; Wadsack, Jörg P.; Wendehals, Lothar; Welsh: Towards Pattern-Based Design Recovery. In: Proc. of the 24th International Conference on Software Engineering (ICSE), Orlando, Florida, USA, ACM Press, Mai 2002, S. 338–348

[NWW01] Niere, J.; Wadsack, J. P.; Wendehals, L.: Design pattern recovery based on source code analysis with fuzzy logic, Technical Report tr-ri-01-222, University of Paderborn, 2001

[NWW03] Niere, Jörg; Wadsack, Jörg P.; Wendehals, Lothar: Handling Large Search Space in Pattern-Based Reverse Engineering. In: Proc. of the 11th International Workshop on Program Comprehension (IWPC), Portland, USA, IEEE Computer Society Press, Mai 2003, S. 274–279

[OO84] Ottenstein, Karl J.; Ottenstein, Linda M: The program dependence graph in a Software development environment. In: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments. 1984, pp. 177-184.

[Pet05] Petterson, Niklas: Measuring Precision for static and dynamic design pattern recognition as a function of coverage. In: International Conference on Software Engineering, Proceedings of the third international workshop on dynamic analysis, Missouri, 2005

[PP94] Paul, S.; Prakash, A.: A Framework for Quellcode Search using Program Patterns, IEEE Transactions on Software Engineering, Bd. 20, 1994, S. 463-475

[Qay10] Qayum, Fawad: Automated assistance for search-based refactoring using unfolding of graph transformation systems. In: Proceedings of the 5th International Conference on Graph Transformations, Springer Verlag, Berlin, 2010

[Rij97] Rajlich, V.: A model for change propagation based on graph rewriting. In: Proc. Int'l Conf. Software Maintenance IEEE Computer Society, 1997, S. 84–91

[Roz97] Rozenberg, Grzegorz (Hrsg.): Handbook of Graph Grammars and Computing by Graph Transformation. Bd. 1. World Scientific, 1997

[SO06] Shi, N.; Olsson, R.A.: Reverse engineering of design patterns from java source ode. In: 21st IEEE/ACM International Conference on Automated Software Engineering, 2006

[SSL01] Simon, F.; Steinbrückner, F., Lewerentz, C.:Metrics based refactoring. In Proc. European Conf. Software Maintenance and Reengineering, IEEE Computer Society, 2001, S. 30–38

[TDB11] Travkin, O.; von Detten, M.; Becker, S.: Towards the Combination of Clustering-based and Pattern-based Reverse Engineering Approaches. In Proceedings of the 3rd Workshop of the GI Working Group L2S2 - Design for Future 2011, Karlsruhe, Germany, Februar 2011

[TBR06] Truong, Nghi; Bancroft, Peter; Roe, Paul: A Web Environment for Learning to Program. In: Transforming IT education: Promoting a culture of excellence. Bruce, C.S.; Mohay, G.; Smith, G.; Stoodley, I.; & Tweedale, R. (Eds.). Santa Rosa, California: Informing Science Press, 2006

[Tra06] Travkin, Dietrich: Bewertung automatisch erkannter Instanzen von Software-Mustern, Institut für Informatik, Diplomarbeit, Universität Paderborn, August 2006.

[Tra07] Travkin, Dietrich: Bewertung automatisch erkannter Ausprägungen von Software-Mustern. In Proc. of the Software Engineering 2007 Conference - Workshop Contributions, Hamburg, Germany, 27.-30.3.2007 (Wolf-Gideon Bleek, Henning Schwentner, and Heinz Züllighoven, eds.), Bd. P-106 of *LNI*, Gesellschaft für Informatik, März, 2007, S. 369-372

[TRB04] Truong, Nghi; Roe, Paul; Bancroft, Peter: Static Analysis of Students' Java Programs. In: Proc. Sixth Australasian Computing Education Conference CRPIT, Bd. 30.R. and Young, A. L., Eds., Dunedin, New Zealand, 2004, S. 317-325

[Wal03] Neil Walkinshaw: The Java System Dependence graph, Technical Report EFoCS-46-2003, Glasgow, 2003

[War62] Warshall, S.: A Theorem on Boolean Matrices. In: ACM Journal, 1962, S. 11-12.

[Wei81] Weiser, Marc: Program slicing. In: Proc. 5th Int. Conference on Software Engineering, IEEE, 1981, S. 439-449

[Wen05] Wenzel, Sven, Automatic Detection of Incomplete Instances of Structural Patterns in UML Class Diagrams. In: Proc. Of the 3rd Nordic Workshop on UML and Software Modeling, Tampere, Finland, Universität Tampere, August 2005

[WH07] Heuer, Tim; Weiß, Raphael: Generierung eines Systemabhängigkeitsgraphen, Bachelorarbeit, Universität Duisburg-Essen, 2007

[Wie04] Wiesmann, André: Über die Abbildung von Java-Programmen auf Graph-Strukturen, Diplomarbeit, Universität Duisburg-Essen, 2004

[WWR03] Wood, Murray; Walkinshaw, Neil; Roper, Marc: "The Java system dependence graph", Third IEEE International Workshop on Source Code Analysis and Manipulation, S. 55, 2003

[XML14] Homepage des W3C zum XML Standard, http://www.w3.org/XML/, zuletzt besucht 2014

[Zh98] Zhao, Jianjun: Applying Program Dependence Analysis to Java Software. Proc. Workshop on Software Engineering and Database Systems, 1998 International Computer Symposium, Tainan, TAIWAN, December 1998, S. 162-169

11. Anhang

Im Anhang sind vier Kapitel aufgeführt, welche die Arbeit um folgende Punkte ergänzen:

1. Syntax des Java Code Graphen: In diesem Kapitel wird die Syntax des JCGs beschrieben, d.h. es werden sowohl die zur Verfügung stehenden Knoten- und Kantentypen inkl. ihrer zugehörigen Attribute dargestellt, als auch pro Knotentyp die erlaubten ausgehenden und eingehenden Kantentypen aufgeführt.

2. Unterstützende Graphregeln: Im Hauptteil der Arbeit werden nur die Regeln dargestellt, welche die wichtigsten Transformationsschritte im Rahmen des Ableitungsprozesses eines JPL-Musters und in der Mustersuche ausführen. Für die vollständige Implementierung dieser Prozesse im Rahmen der Anwendungsbeispiele sind aber noch weitere unterstützende Regeln notwendig, die z.B. vorgegebene Bereiche markieren, oder einen Graph so verändern das eine direkte Zuordnung von AJSDG und JCG Knoten vorgenommen werden kann. Diese Regeln werden in Kapitel 11.2. dargestellt. Verweise auf diese Regeln erfolgen sowohl aus dem Hauptteil, als auch aus Kapitel 11.3.

3. Regelzuordnung zwischen AGG-Regeln im praktischen Beispiel und der Regeldarstellung in der Dissertation: Zur Realisierung der in Kapitel 8. beschriebenen Anwendungsbeispiele wurden verschiedene Regelsätze im AGG Tool implementiert. Die verwendeten Regelsätze und Graphen sind auf der DVD enthalten welche dieser Arbeit beiliegt. Weiterhin wurden für ausgewählte Beispiele Videos, welche die automatisierte Ableitung der JPL Muster und den automatisierten Ablauf der Mustersuche zeigen, hinzugefügt.

   In diesem Kapitel wird die Verbindung zwischen den in AGG für die Anwendungsbeispiele implementierten Regeln und Regelabläufen zu den Regelsätzen in dieser Arbeit hergestellt. Hierfür werden für ausgewählte Beispiele die implementierten Regeln tabellarisch aufgelistet und auf die entsprechenden Regeln entweder im Hauptteil der Arbeit, oder in den unterstützenden Regeln verwiesen.

4. In diesem Kapitel wird der Inhalt der beiligenden DVD aufgeführt. Dieser umfasst die Softwareartefakte, welche für Durchführung der in Kapitel 8. beschreibenen Anwendungsbeispiele verwendet wurden. Weiterhin sind Filme enthalten, welche die Ableitung des JPL-Graphen und den Ablauf der Mustersuche für verschiedene Beispiele darstellen.

11.1. Syntax des Java Code Graphen

Im Folgenden aufgeführt sind die Knoten des JCG, sowie die von diesen Knoten ausgehende Kanten. Die Realisierung dieser Konzeption erfolgte im Tool *Java2ggx,* welches Java Quellcode in die im Folgenden dargestellte Struktur transformiert. Die Darstellung ist der technischen Beschreibung des Tools, erstellt von Jens Bartelheimer, entnommen.

Es wird der jeweilige Knotentyp zusammen mit seinen zugehörigen Attributen dargestellt. Nachfolgend werden tabellarisch die ausgehenden Kanten zusammen mit ihren Attributen und den möglichen Zielknotentypen aufgeführt.

project

Attribute:
ausgehende Kanten:

Name	Ziele	Anzahl	Attribute
file	File	1...n	position

file

Attribute:
name String

ausgehende Kanten:

Name	Ziele	Anzahl	Attribute
package	Package	0...1	
include	Include	0...n	
interface	Interface	0...n	

class

Attribute:
abstract boolean
visibility string
name string
final false

ausgehende Kanten:

Name	Ziele	Anzahl	Attribute
implements	Interface BlackBox	0...n	
extends	Class BlackBox	0...1	
classMethod	method	0...n	position
memberVariable	primitiveDeclaration objectDeclaration primitiveArrayDeclaration objectArrayDeclaration	0...n	position

classConstructor	constructor	0...n	position
staticConstructor	staticConstructor	0...1	position
nestedClass	Class	0...n	position
commentBegin	commentBegin	0...1	
commentEnd	comment	0...1	

anonymousClass

Attribute:
-
ausgehende Kanten:

Name	Ziele	Anzahl	Attribute
extends	class BlackBox	0...1	
extends	Class BlackBox	0...1	
classMethod	method	0...n	position
memberVariable	primitiveDeclaration objectDeclaration primitiveArrayDeclaration objectArrayDeclaration	0...n	position
classConstructor	constructor	0...n	position
nestedClass	Class	0...n	position
commentBegin	commentBegin	0...1	
commentEnd	comment	0...1	

accessToAnArrayElement

Attribute:
-
ausgehende Kanten:

Name	Ziele	Anzahl	Attribute
index	Expression	0...n	dimension
arrayDeclaration	primitiveArrayDeclaration objectArrayDeclaration	1	
interface	interface	0...n	

referenceArrayCast

Attribute:
dimension int
class String

ausgehende Kanten:

Name	Ziele	Anzahl	Attribute
class	Class interface BlackBox	1	

explicitArrayInitialization

Attribute:
-
ausgehende Kanten:

Name	Ziele	Anzahl	Attribute
indexAssignment	Expression	1..n	index

arrayPrimitiveTypeInitialization

Attribute:
type String
dimension int
ausgehende Kanten:

Name	Ziele	Anzahl	Attribute
dimensionLength	Expression	0..n	dimension
assignment	explicitArrayInitialization	0..1	

arrayReferenceTypeInitialization

Attribute:
class String
dimension int
ausgehende Kanten:

Name	Ziele	Anzahl	Attribute
dimensionLength	Expression	1..0	dimension
assignment	explicitArrayInitialization	0..1	
class	Class interface BlackBox	1	

constructorChaining

Attribute:
superclassConstructor boolean

ausgehende Kanten:

Name	Ziele	Anzahl	Attribute
parameterDeclaration	Expression	0..n	position
constructorInvocation	constructor	0..1	
comment	comment	0..1	

switch

Attribute:
label String

ausgehende Kanten:

Name	Ziele	Anzahl	Attribute
case	case	0..n	position
default	default	0..1	position
expression	expression	0..1	
lock	begin	1	

default
(SimpleNode)

Attribute:
-
ausgehende Kanten:

Name	Ziele	Anzahl
block	begin	1

case
Attribute:
-
ausgehende Kanten:

Name	Ziele	Anzahl
constant	constant	1
block	begin	1

break

Attribute:
label String

ausgehende Kanten:

Name	Ziele	Anzahl
break	end	1
comment	comment	0...1

finally

Knoten für finally-Abschnitt eines try-catch-Blockes

Attribute:
-
ausgehende Kanten:

Name	Ziele	Anzahl
block	begin	1

statement

Knoten für leeres Statement: ;

Attribute:
-
ausgehende Kanten:

Name	Ziele	Anzahl
comment	comment	0...1

continue

Attribute:
label String

ausgehende Kanten:

Name	Ziele	Anzahl
continue	For While do-while	1
comment	comment	0...1

assignment

Knoten für alle Zuweisungsoperatoren (=, -=, +=, ...)

Attribute:
name String

ausgehende Kanten:

Name	Ziele	Anzahl
leftOperator	Expression*	1
rightOperator	Expression*	1
comment	comment	0...1

increment / decrement

Attribute:
type String (posfix|prefix)

ausgehende Kanten:

Name	Ziele	Anzahl
rightHandSide oder leftHandSide (je nach Typ)	Expression*	1
comment	comment	0...1

primitiveDeclaration

Attribute:
accessLevel String
static boolean
final boolean
transient boolean
volatile boolean
name String
type String

ausgehende Kanten:

Name	Ziele	Anzahl	Bemerkung
assignment	Expression*	0...1	KantennichtbeiParametern
comment	comment	0...1	

objectDeclaration

Attribute:
accessLevel String
static boolean
final boolean

transient boolean
volatile boolean
identifier String
class String

ausgehende Kanten:

Name	Ziele	Anzahl	Bemerkung
objectDeclaration	Interface Anonymousclass Class BlackBox include	1	Bemerkung
assignment	Expression	0..1	Kanten nicht bei Parametern
comment	comment	0..1	

primitiveArrayDeclaration

Attribute:
accessLevel String
static boolean
final boolean
transient boolean
volatile boolean
name String
type String
dimension int

ausgehende Kanten:

Name	Ziele	Anzahl	Bemerkung
assignment	Expression*	0...1	Kanten nicht bei Parametern
comment	comment	0...1	

objectArrayDeclaration

Attribute:
accessLevel String
static boolean
final boolean
transient boolean
volatile boolean
identifier String
class String
dimension int

ausgehende Kanten:

Name	Ziele	Anzahl	Bemerkung
objectDeclaration	interface,anonymousclass,class,BlackBox		
include	1		
assignment	Expression*	0...1	Nicht bei

			Parametern
comment	comment	0...1	

cast

Attribute:
-
ausgehende Kanten:

Name	Ziele	Anzahl
definition	Class Interface primitiveArrayCast referenceArrayCast BlackBox	1
rightHandSide	Expression	Anzahl

catch

Attribute:
-
ausgehende Kanten:

Name	Ziele	Anzahl	
block	begin	1	
catchException	object	declaration	1

constructor

Attribute:
accessLevel String

ausgehende Kanten:

Name	Ziele	Anzahl	Attribut
parameterDeclaration	objectDeclaration primitiveDeclaration primitiveArrayDeclaration objectArrayDeclaration	0..n	position
exception	throwexception	0...n	position
constructorChaining	constructorChaining	0..1	
body	begin	1	

ifthen

Attribute:
label String

ausgehende Kanten:

Name	Ziele	Anzahl	Attribut
if	If else..if	0..n	position
else	else	0..1	

if, elseif, else

Attribute:
-

ausgehende Kanten:

Name	Ziele	Anzahl	Attribut
Block	Begin	1	
Expression	Expression	1 (nur bei if und esleif	

include

Attribute:
name String

ausgehende Kanten:

Name	Ziele	Anzahl
packageImport	Package packageSubdirecktory	1
comment	comment	0..1

staticConstructor

Attribute:
static boolean

ausgehende Kanten:

Name	Ziele	Anzahl
block	begin	1
comment	comment	0...2

interface

Attribute:
abstract boolean
visibility string
name string

ausgehende Kanten:

Name	Ziele	Anzahl	Attribut
extends	Interface BlackBox	0..n	Attribut
interfaceMethod	method	0..n	position
memberVariable	primitiveDeclaration objectDeclaration primitiveArrayDeclaration objectArrayDeclaration	0..n	position
commentBegin	comment	0..1	
commentEnd	comment	0...1	

for

Attribute:
label String

ausgehende Kanten:

Name	Ziele	Anzahl
initialization	Expression*	0...n
terminiation	Expression*	0...1
expression	Expression*	0...n
block	begin	1

while, do-while

Attribute:
label String

ausgehende Kanten:

Name	Ziele	Anzahl
terminiation	Expression*	0...1
block	begin	1

methodCall

Attribute:
name String

ausgehende Kanten:

Name	Ziele	Anzahl	Attribut
parameter	Expression*	0...n	position
methodDefinition	method	0...1	

method

Attribute:
static boolean
accessLevel boolean
abstract boolean
final boolean
native boolean
synchronizied boolean
returnType String
name String

ausgehende Kanten:

Name	Ziele	Anzahl	Attribut	Bemerkung
parameterDeclaration	primtiveDeclaration objectDeclaration primitiveArrayDeclaration objectArrayDeclaration	0..n	position	
exceptions	throwexception	0...n	position	Bemerkung
signOver	method	0...1		
implements	method	0...1	Attribut	Kante zu implementierter, abstrakten Methode der Superklasse

implements	method	0...1		Kante zu implementierter Methode eines Interfaces
Body	begin	0...1		bei abstrakter Methode nicht vorhanden, da dort kein Body vorhanden ist

objectInitialization

Attribute:
class String

ausgehende Kanten:

Name	Ziele	Anzahl	Attribut
objectInitialization class	interface	include	1
constructorInvocation	constructor	0...1	
parameter	Expression*	0...n	position
Class	anonymousClass	0...1	
comment	comment	0...1	

package

Attribute:
name String

ausgehende Kanten:

Name	Ziele	Anzahl
subdirectory	packageSubdirectory	0...1
comment	comment	0...1

return

Attribute:
-

ausgehende Kanten:

Name	Ziele	Anzahl
expression	Expression*	0...1
comment	comment	0...1
return	methodCall	0…n

chaining

(SimpleTreeNode)

Attribute:
-

ausgehende Kanten:

Name	Ziele	Anzahl
leftHandSide	Expression*	1
rightHandSide	Expression*	1
comment	comment	0...1

instanceof

(SimpleTreeNode)

Attribute:
-

ausgehende Kanten:

Name	Ziele	Anzahl
leftHandSide	Expression*	1
rightHandSide	Expression*	1

***, /, mod, ...**

(SimpleTreeNode)

Attribute:
-

ausgehende Kanten:

Name	Ziele	Anzahl
leftHandSide	Expression*	1
rightHandSide	Expression*	1

( )
(SimpleTreeNode)

Attribute:
-

ausgehende Kanten:

Name	Ziele	Anzahl
contains	Expression*	1
comment	comment	0...1

synchonized

Attribute:
-

ausgehende Kanten:

Name	Ziele	Anzahl
Reference	Expression*	0...1
Block	begin	1

this

Attribute:
-
ausgehende Kanten:

Name	Ziele	Anzahl
Declaration	class	1

super

Attribute:
-
ausgehende Kanten:

Name	Ziele	Anzahl
Declaration	BlackBox class	1

throwexception

Attribute:
class String

ausgehende Kanten:

Name	Ziele	Anzahl	Bemerkung
instance	Expression	1	nur bei throw-Statement (throw ...)
classReference	Class include	1	nur bei Methodendefinition (throws ...)
comment	comment	0..1	

try

Attribute:
-
ausgehende Kanten:

Name	Ziele	Anzahl	Attribut
block	begin	1	
Catch	catch	1...n	position
finally	finally	0...1	

accessToPrimitiveTypeVariable

Attribute:
name String

ausgehende Kanten:

Name	Ziele	Anzahl
variableDeclaration	primitivedeclaration	1

accessToReferenceTypeVariable

Attribute:
name String

ausgehende Kanten:

Name	Ziele	Anzahl
variableDeclaration	objectDeclaration	1

accessToArray

Attribute:
name String

ausgehende Kanten:

Name	Ziele	Anzahl
variableDeclaration	arrayDeclaration	1

accessToStaticReference

Attribute:
name String

ausgehende Kanten:

Name	Ziele	Anzahl
variableDeclaration	Class Interface include	1

constant

Attribute:
content String

literal

Attribute:
string String

false, true

comment

Attribute:
name String

ausgehende Kanten

Name	Ziele	Anzahl
comment	comment	0...1

begin, end

Attribute:
ausgehende Kanten:

Name	Ziele	Anzahl
Comment	Comment	0...1

BlackBox
Attribute:
name String

ausgehende Kanten:

Name	Ziele	Anzahl
Declaration	Include	1

consecutive - Kanten

Knoten innerhalb eines Anweisungsblockes werden über „consective“-Kanten verbunden. Die „consecutive“-Kanten beginnen beim „begin“-Knoten des Blockes, verbinden die Knoten des Blockes und enden im „end“-Knoten des Blockes. Falls darauf ein erneuter Block folgt, wird eine Kante vom „end“-Knoten zum „begin“-Knoten des darauffolgenden Blockes gezogen.

Expression:
Hinter dem Ziel Expression können sich verschiedene, zusammengesetzte Ausdrücke verbergen. Zum einen kann sich hier ein einfacher „accessToReferceneNode“ verstecken, zum anderen aber auch komplexe Ausdrücke wie ein Methodenaufruf mit Parametern und Cast, die über chaining-Knoten verbunden sind.

11.2. Unterstützende Graphregeln

Regeln zur Transformation des Quelltextgraphen im Ablauf der Suche

Bereichsmarkierung für Bereiche innerhalb von Klassen

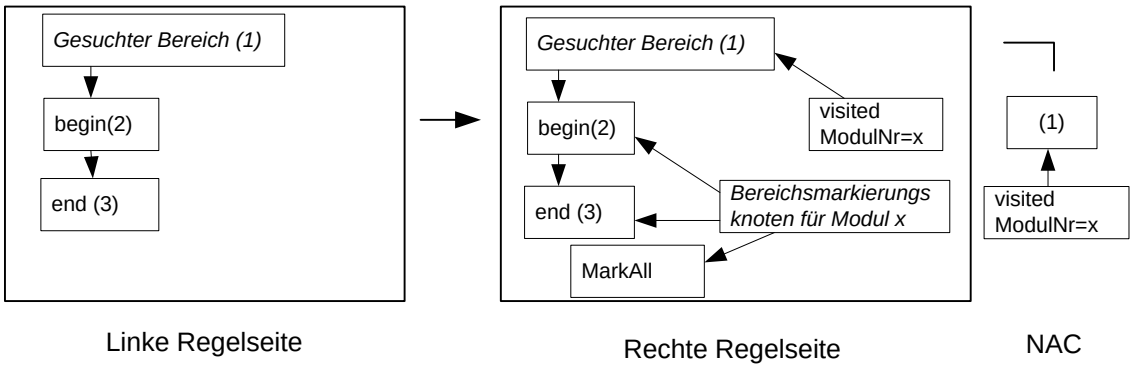


Abbildung 87: Bereichsmarkierung für Bereiche innerhalb von Klassen 1

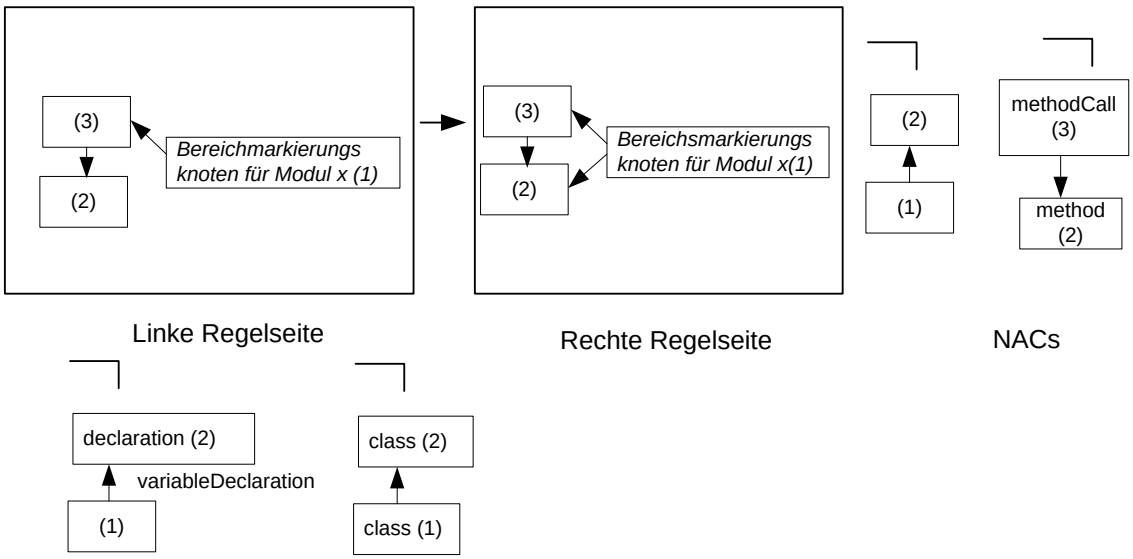


Abbildung 88: Bereichsmarkierung für Bereiche innerhalb von Klassen 2

Der Knoten des Typs *Gesuchter Bereich* muss mit dem Element das den Gültigkeitsbereich in der Identifikatorsektion des Moduls beschreibt ersetzt werden. Das Attribut *ModulNr* der Knoten vom Typ *visited* und der *Bereichsmarkierungsknoten* muss parametrisiert werden.

Bedingungen für den Regelsatz:

Vorbedingung: Im Quelltextgraph existiert der gesuchte Bereichsknoten (*while, method,* etc.). Weiterhin wurde der Bereich noch nicht durch den *visited* Knoten markiert.
Nachbedingung: Alle Knoten innerhalb des Bereichs sind mit einem Bereichsmarkierungsknoten verbunden.

Bereichsmarkierung für Klassen

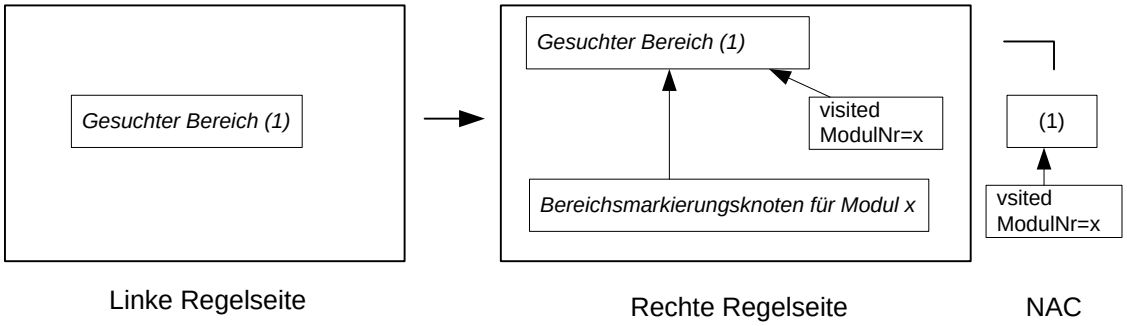


Abbildung 89: Bereichsmarkierung für Klassen 1

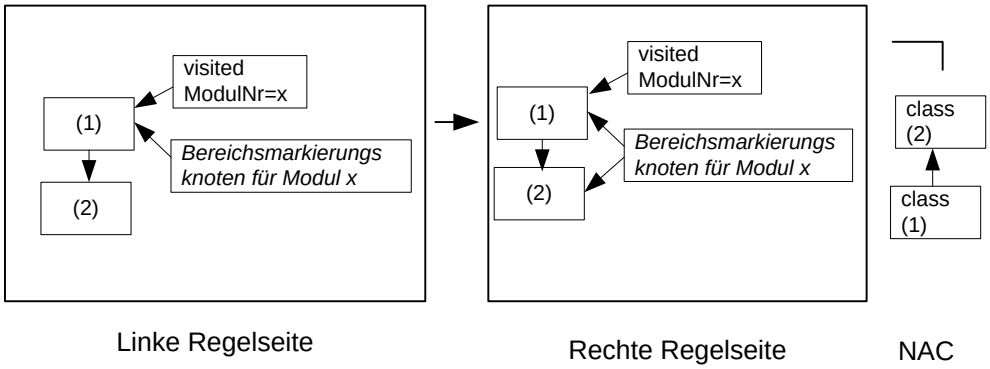


Abbildung 90: Bereichsmarkierung für Klassen 2

Der Knoten des Typs *Gesuchter Bereich* muss mit dem Element das den Gültigkeitsbereich in der Identifikatorsektion des Moduls beschreibt ersetzt werden. Der zu markierende Bereich wird durch einen *class* Knoten beschrieben. Das Attribut *ModulNr*, der Knoten vom Typ *visited* und der *Bereichsmarkierungsknoten* müssen parametrisiert werden.

Bedingung für den Regelsatz:

Vorbedingung: Es existiert ein noch nicht besuchter Klassenknoten.
Nachbedingung: Der Knoten vom Typ *class* und alle diesem direkt nachfolgenden Knoten wurden mit einem *Bereichsmarkierungsknoten* verbunden. Ausgenommen sind hier weitere Knoten vom Typ *class* (Vererbungsstruktur).

Nachfolgend wird die Regel aus Abbildung 88 angewendet.

Bereichsmarkierung mit hierarchischer Abhängigkeit

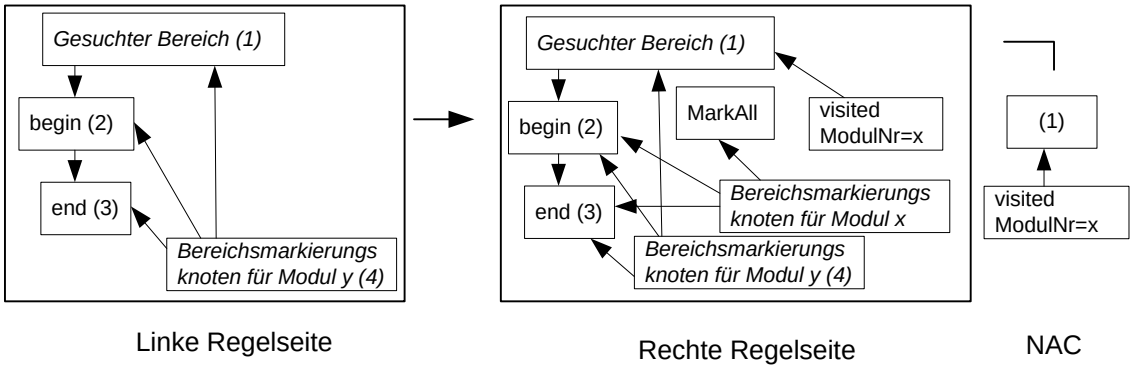


Abbildung 91: Bereichsmarkierung mit hierarchischen Abhängigkeiten

Der Knoten des Typs *Gesuchter Bereich* muss mit dem Element das den Gültigkeitsbereich in der Identifikatorsektion des Moduls beschreibt ersetzt werden. Das Attribut *ModulNr*, der Knoten vom Typ *visited* und der Bereichsmarkierungsknoten muss für die Module x und y parametrisiert werden.

Bedingungen für den Regelsatz:

Vorbedingung: Es existiert ein noch nicht besuchter Gültigkeitsbereich. Die zugehörigen Knoten vom Typ *begin* und *end* wurden bereits markiert, d.h. sie gehören zum Bereich eines hierarchisch höherstehenden Moduls y.
Nachbedingung: Der *begin* und *end* Knoten wurden mit einem *Bereichsmarkierungsknoten* für Modul x, d.h. dem hierarisch unter Modul y stehendem Modul markiert.

Nachfolgend wird die Regel aus Abbildung 88 angewendet.

Bereichsmarkierung für AJSDG-Knoten

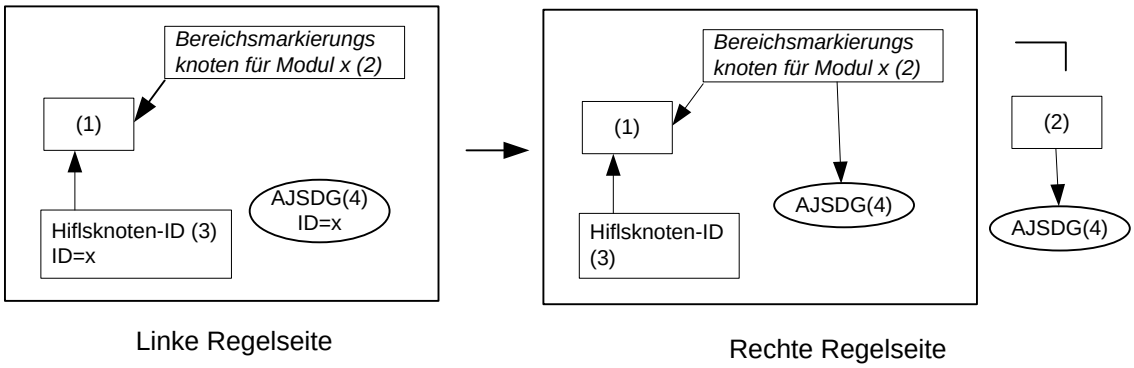


Abbildung 92: Bereichsmarkierung für ASDG-Knoten

Markierung der nicht zu den Bereichen gehörenden Knoten

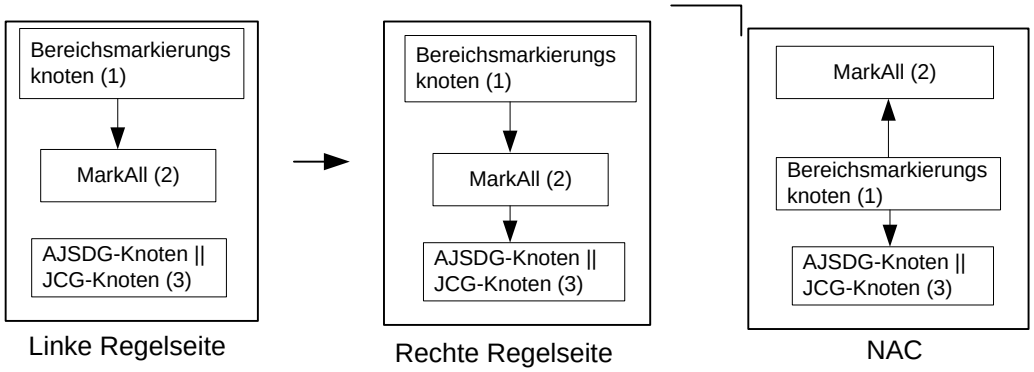


Abbildung 93: Markierung nicht zum Bereich gehörender Knoten mit Bereichsmarkierungsknoten

Alternative Regel zur Markierung der nicht zum Bereich gehörenden Knoten:

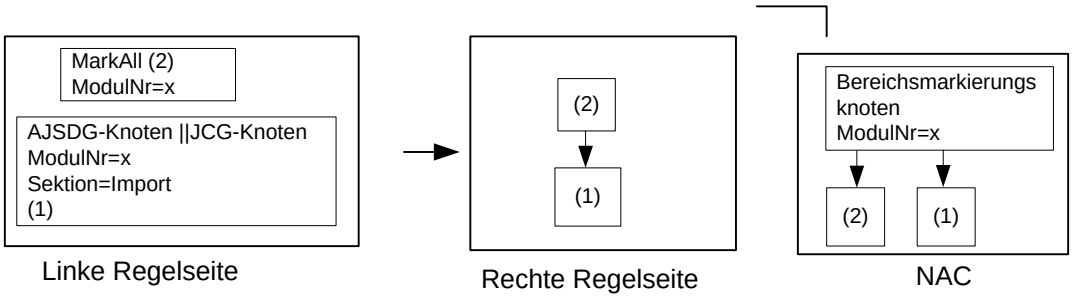


Abbildung 94: Markierung nicht zum Bereich gehörender Knoten über Attribut *ModulNr*

Markierung möglicher Quellknoten für die Pfadsuche

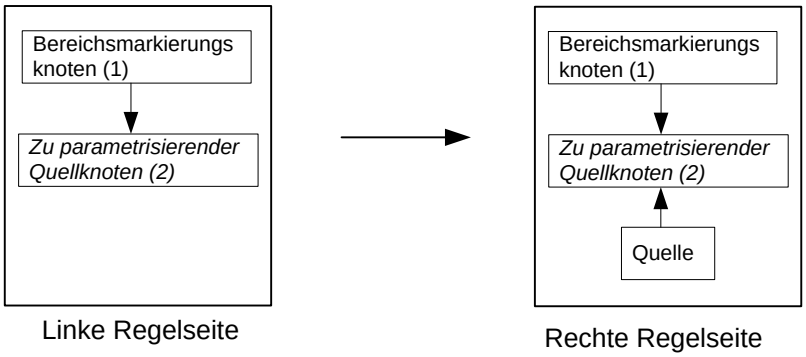


Abbildung 95: Markierung möglicher Quellknoten für die Pfadsuche

Markierung möglicher Zielknoten für die Pfadsuche

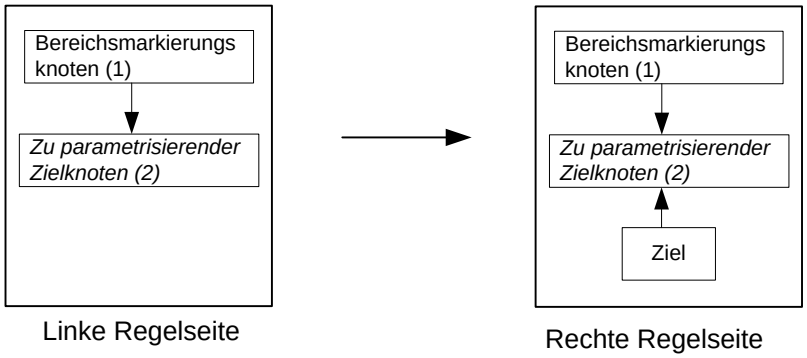


Abbildung 96: Markierung möglicher Zielknoten für die Pfadsuche

Regeln zur Transformation des JPL-Graphen

Markierung der Knoten eines Moduls x mit dem Bereichsmarkierungsknoten, der zuvor erstellt wurde

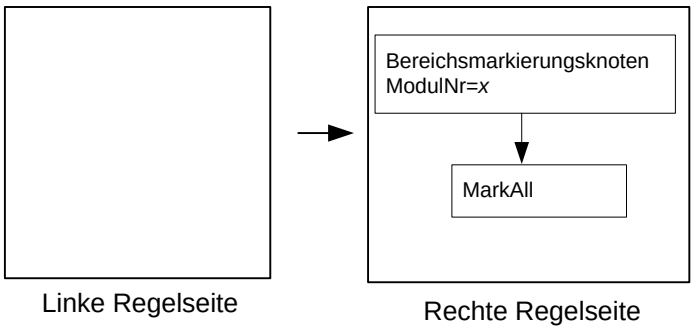


Abbildung 97: Initiale Erstellung eines Bereichsmarkierungsknotens mit Verbindung zum *MarkAll*-Knoten

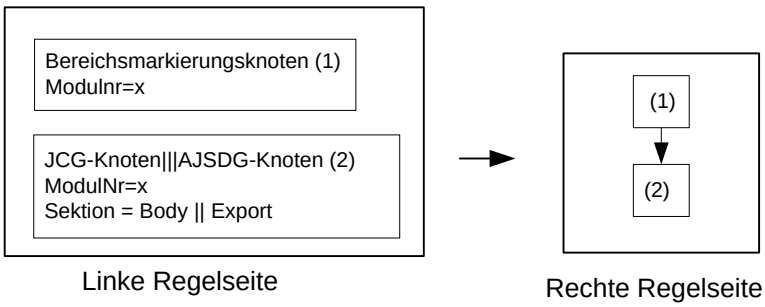


Abbildung 98: Markierung der JCG/AJSDG-Knoten in einem Modul

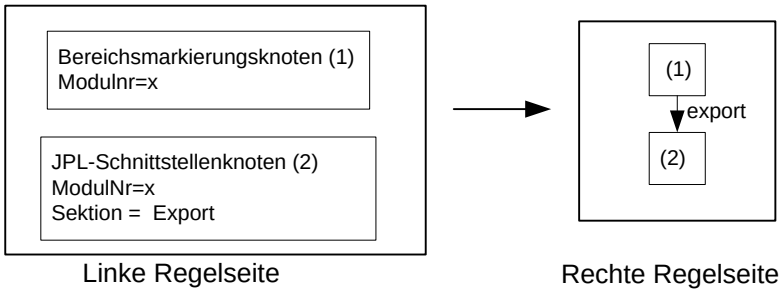


Abbildung 99: Markierung der JPL-Schnittstellenknoten: direkte Übergabe 1

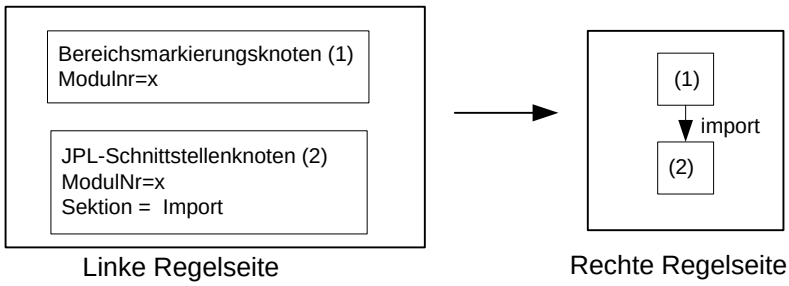


Abbildung 100: Markierung der JPL-Schnittstellenknoten: direkte Übergabe 2

Markierung für die Variante „indirekte Variablenübergabe durch einen return-Rückgabewert“

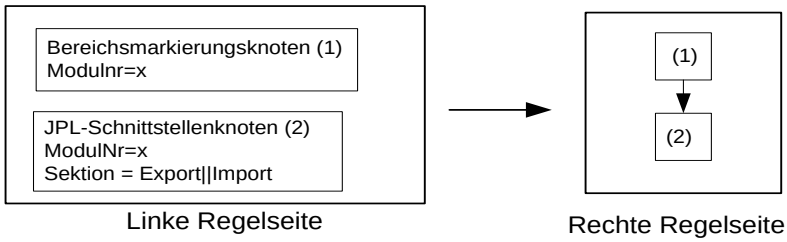


Abbildung 101: Markierung der JPL-Schnittstellenknoten: indirekte Übergabe

Markierung der Knoten, die zu keinem Bereich gehören dürfen

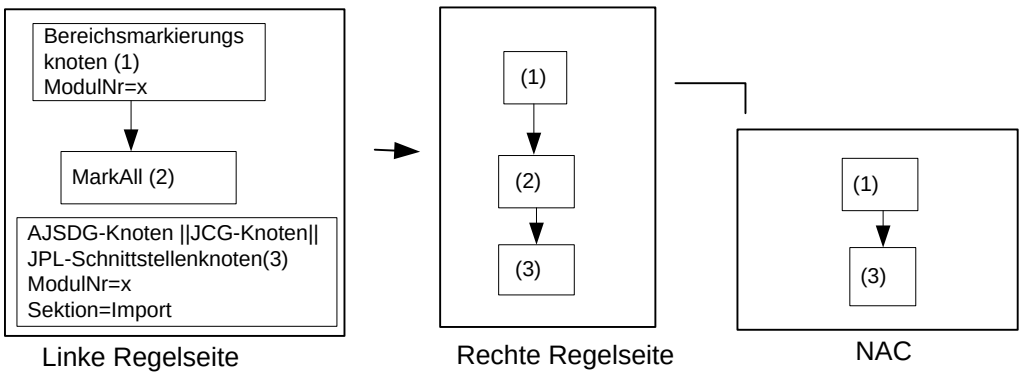


Abbildung 102: Markierung der Knoten außerhalb des Gültigkeitsbereichs 1

In der zuvor dargestellten Regel erfolgt die Zuordnung der zu markierenden Knoten über das Attribut *ModulNr* des Bereichsmarkierungsknotens. Unten ist die Zuordnung direkt über den *MarkAll*-Knoten realisiert. Die Regeln in Abbildung 102 und Abbildung 103 ähneln stark den Regeln in Abbildung 93 und Abbildung 94. Zu unterscheiden ist hier der Regelkontext und der Bezug zum Attribut *Sektion=Import*.

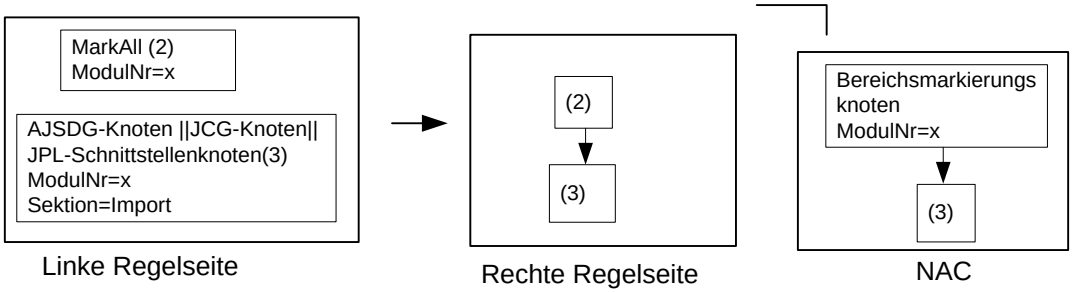


Abbildung 103: Markierung der Knoten außerhalb des Gültigkeitsbereichs 2

Alternativ zur Markierung über den *MarkAll*-Knoten können NACs in die Suchregel eingefügt werden (alternativer Suchalgorithmus). Für jeden Knoten, der nicht zum Modulbereich gehörend darf (Deklaration im Import) wird folgende NAC erstellt:

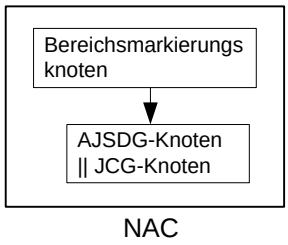


Abbildung 104: NAC-Variante zum *MarkAll*-Knoten

Der *Bereichsmarkierungsknoten* muss auf den entsprechenden Knoten des Moduls auf der linken Regelseite gemapped werden.

Transformation des *JPLPrimitiveDeclaration*- zum *methodCall*-Knoten

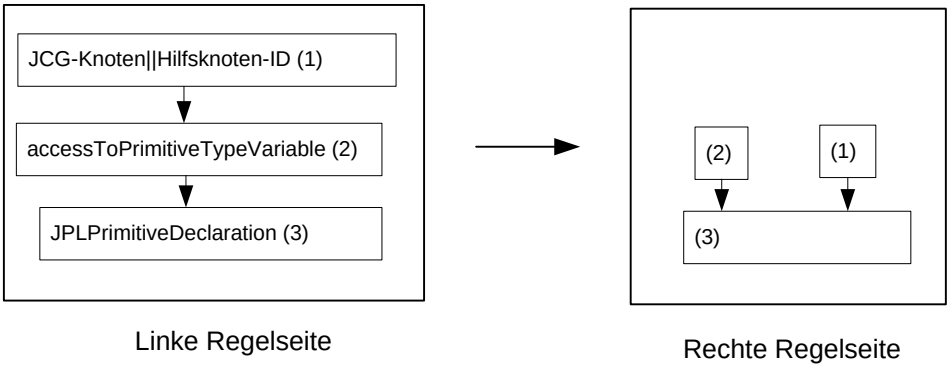


Abbildung 105: Entfernen der Kanten zum *accessToPrimitiveTypeVariable*-Knoten

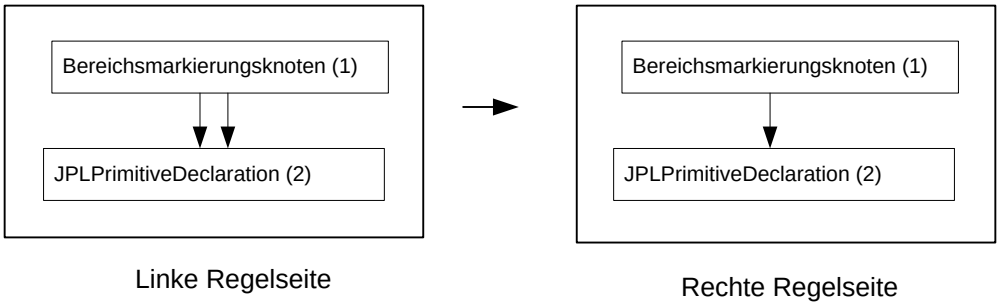


Abbildung 106: Entfernen doppelter Kanten zum *Bereichsmarkierungsknoten*

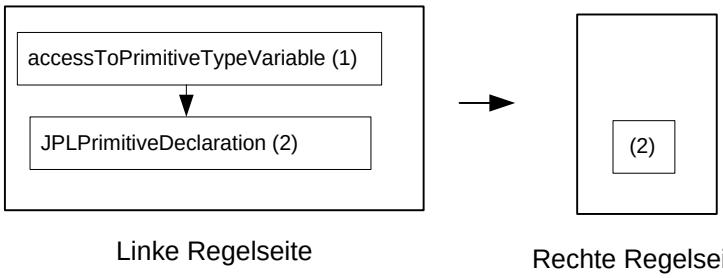


Abbildung 107: Entfernen des *accessToPrimitiveTypeVariable*-Knotens

Bedingungen für den Regelsatz:

Vorbedingung: Ein Knoten vom Typ *accessToPrimitiveTypeVariable* ist im JPL-Graph mit weiteren Knoten verbunden.
Nachbedingung: Die Kanten wurden auf den *JPLPrimitiveDeclaration*-Knoten umgeleitet und der *accessToPrimitiveTypeVariable*-Knoten wurde gelöscht.

Nachfolgend werden Hilfsregeln zur Erstellung der Übergabevariante durch den *return*-Parameter auf dem JPL-Graphen dargestellt.

Falls ein Knoten vom Typ *Hilfsknoten-ID* mit dem *JPLPrimitiveDeclaration*-Knoten verbunden wurde, müssen die beiden nachfolgenden Regeln angewendet werden, um auch Beziehung zwischen JCG- und AJSDG-Knoten in das Suchmuster zu übertragen (der *JPLPrimitiveDeclaration*-Koten durfte zuvor, abweichend zu Abbildung 42, nicht umbenannt werden). Auch alle *JCG-Knoten* und *Bereichsmarkierungsknoten* müssen umgehängt werden.

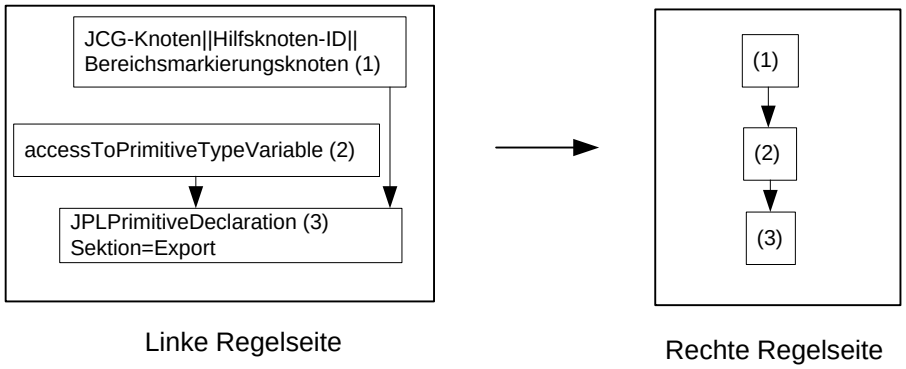


Abbildung 108: Umhängen auf den *JPLPrimitiveDeclaration*-Knoten

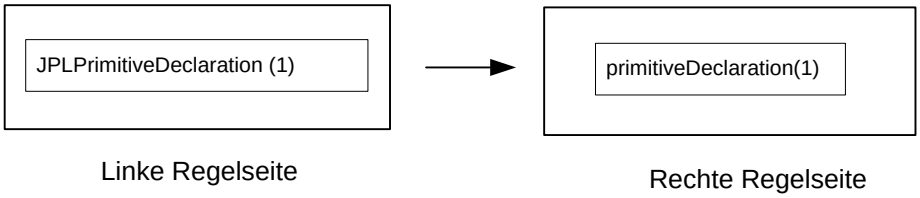


Abbildung 109: Umbenennung der *JPLPrimitiveDeclaration*

Transformation: Ableitung von JCG- und AJSDG-Übergabestrukturen

Ableitung der JCG Übergabestruktur:

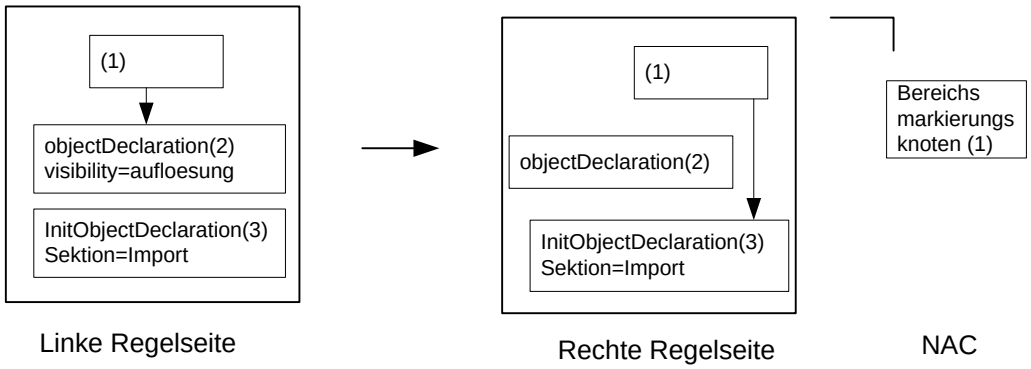


Abbildung 110: Ableitung von direkten JCG-Übergabestrukturen 1

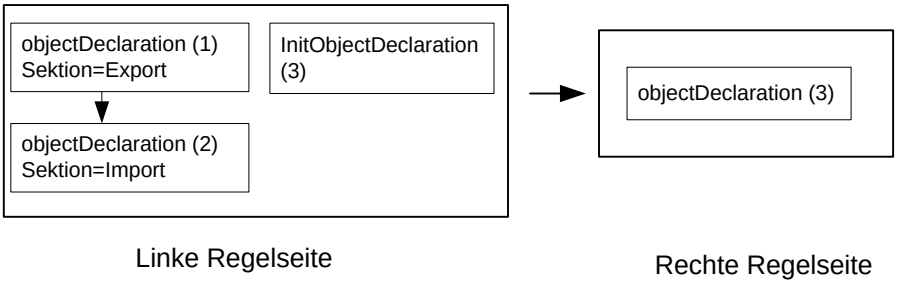


Abbildung 111: Ableitung von direkten JCG-Übergabestrukturen 2

Die mit dem importierenden und exportierenden Knoten verbundenen Knoten werden auf einen neu zu erstellenden *InitObjectDeclaration*-Knoten übertragen und die Übergabestruktur nachfolgend gelöscht. Da der *InitObjectDeclaration*-Knoten zu keinem der beiden Modulbereiche gehört, in welche die Export- und Importknoten eingebettet sind, werden keine Bereichsmarkierungen mit diesem verbunden.

Ableitung der AJSDG Übergabestruktur:

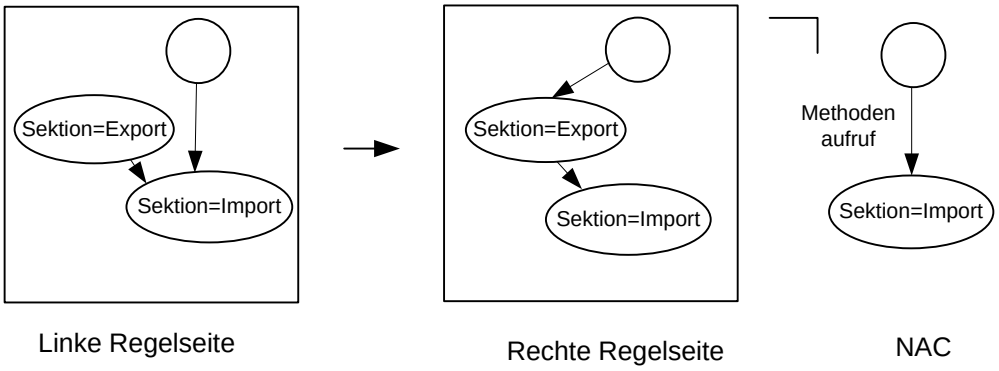


Abbildung 112: Ableitung von direkten AJSDG-Übergabestrukturen 1

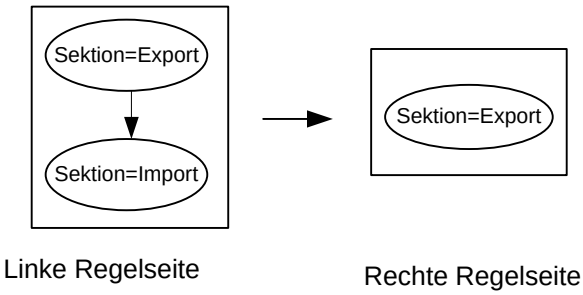


Abbildung 113: Ableitung von direkten Übergabestrukturen 2

Die mit dem importierenden Knoten verbundenen Knoten werden auf den exportierenden übertragen und der importierende Knoten nachfolgend gelöscht. Der Regelsatz wird nur ausgeführt, wenn es sich nicht um einen Methodenaufruf handelt.

Alternative Regelsätze zu Abbildung 38 ff.: Erstellung von Implementierungsvarianten auf dem JPL-Graphen für JPL-Schnittstellenknoten, die mit Knoten im Modulkörper verbunden sind

Folgende Regelsätze erzeugen auf dem JPL-Graphen Implementierungsvarianten zu JPL-Schnittstellenstrukturen. Vorbedingung ist, dass die JPLObjectDeclaration-Knoten mit einem referenzierenden Knoten verbunden sind.

Hinweise:

- In den folgenden Regeln zur indirekten Variablenübergabe wird davon ausgegangen, dass auf beide JPL-Schnittstellenknoten ein Verweis eines Referenzknotens besteht. Es sind auch Fälle möglich, in denen nur auf einen Knoten ein Verweis besteht, oder das mehrere Verweise auf einen Knoten zeigen. Diese Fälle sind im Folgenden nicht berücksichtigt.
- Da die Knoten vom Typ *accessToReferenceVariable* bereits alle mit Bereichsmarkierungsknoten verbunden sind, können diese von den Schnittstellenknoten getrennt werden, ohne neu verbunden werden zu müssen.

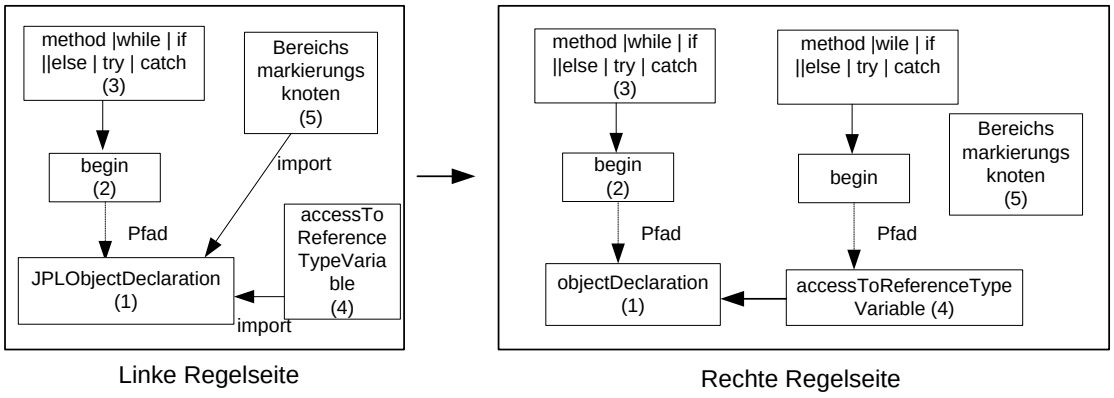


Abbildung 114: Ableitungsvariante für Muster mit verbundenen Referenzknoten 1

Alternativer Regelsatz zu Abbildung 40

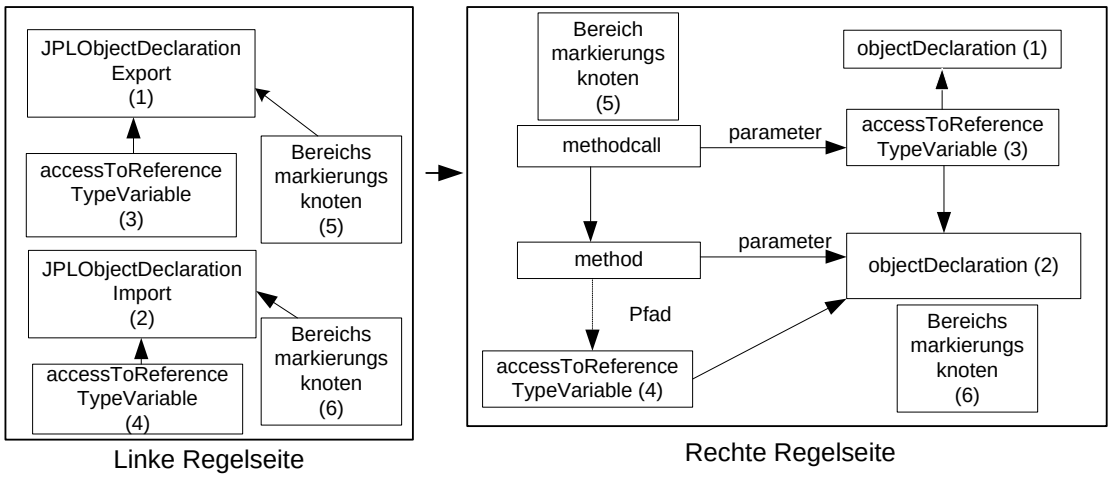


Abbildung 115: Ableitungsvariante für Muster mit verbundenen Referenzknoten 2

Alternativer Regelsatz zu Abbildung 41

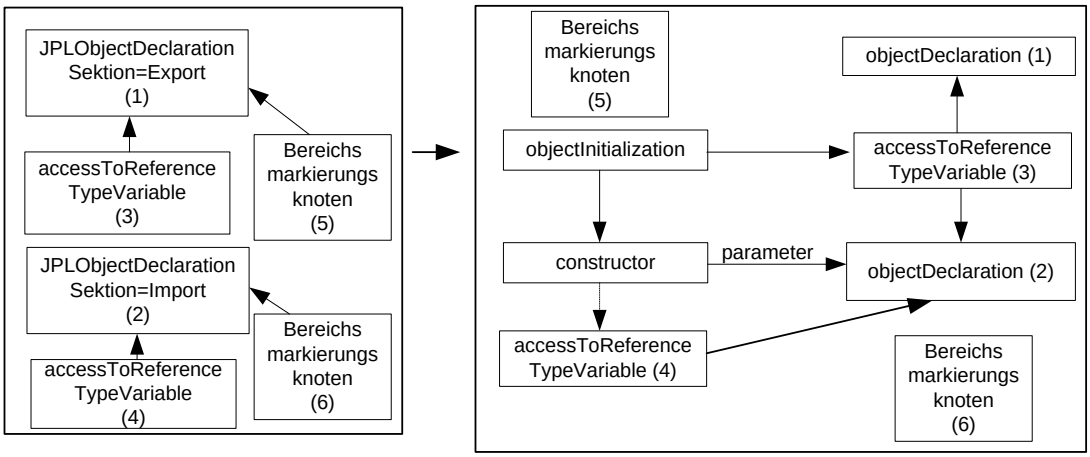


Abbildung 116: Ableitungsvariante für Muster mit verbundenen Referenzknoten 3

Alternativer Regelsatz zu Abbildung 44
Abbildung 44

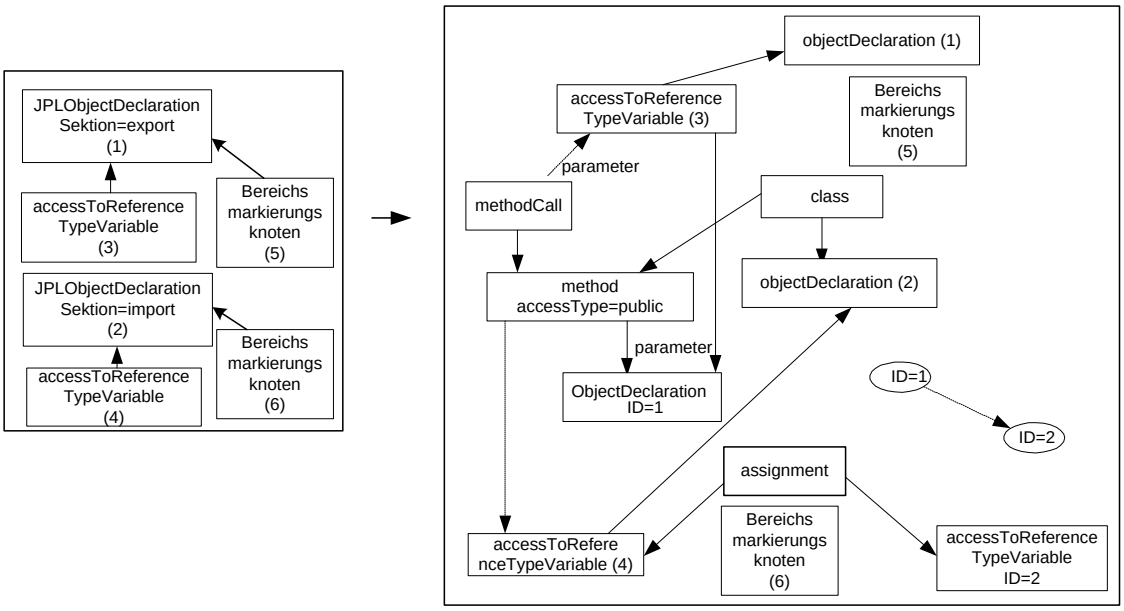


Abbildung 117: Ableitungsvariante für Muster mit verbundenen Referenzknoten 4

Alternativer Regelsatz zu Abbildung 45

Zu beachten: Verbundene Referenzknoten zu den importierenden JPL-Schnittstellenknoten werden in dieser Implementierungsvariante gelöscht, da auf Methodenaufrufe keine Referenzen gerichtet sein können.

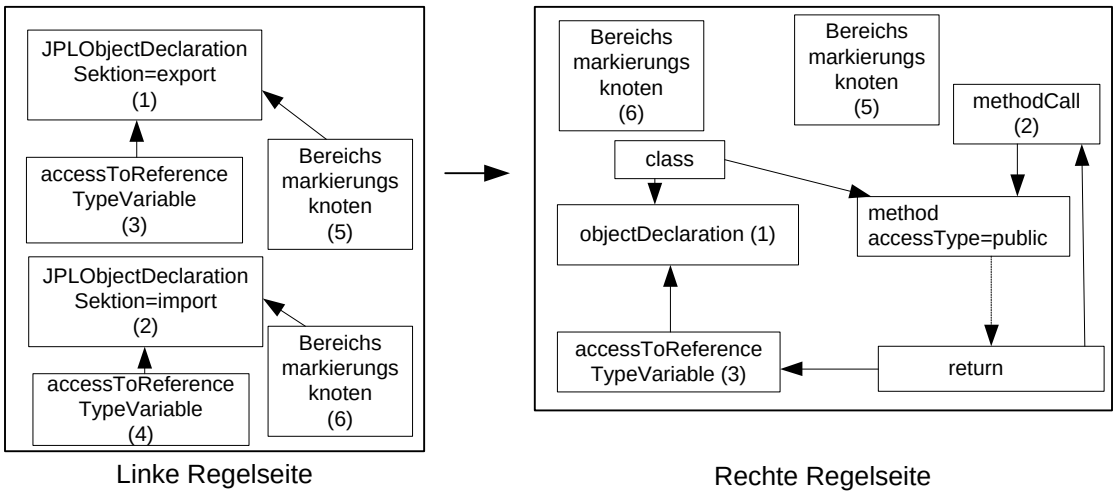


Abbildung 118: Ableitungsvariante für Muster mit verbundenen Referenzknoten 5

Identifikation des Gesamtmusters in der sektionsbasierten Mustersuche

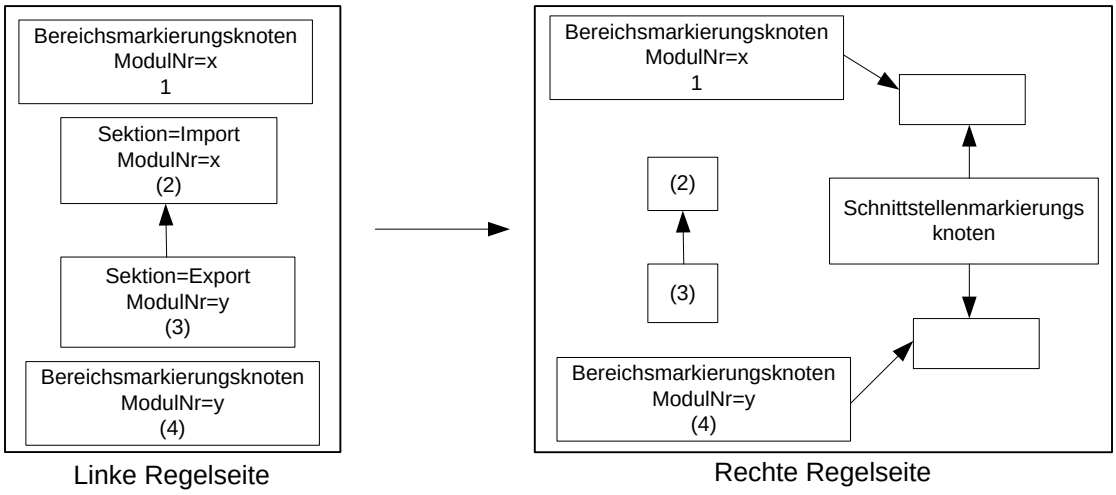


Abbildung 119: Einfügen der Schnittstellenstruktur in das Gesamtmuster

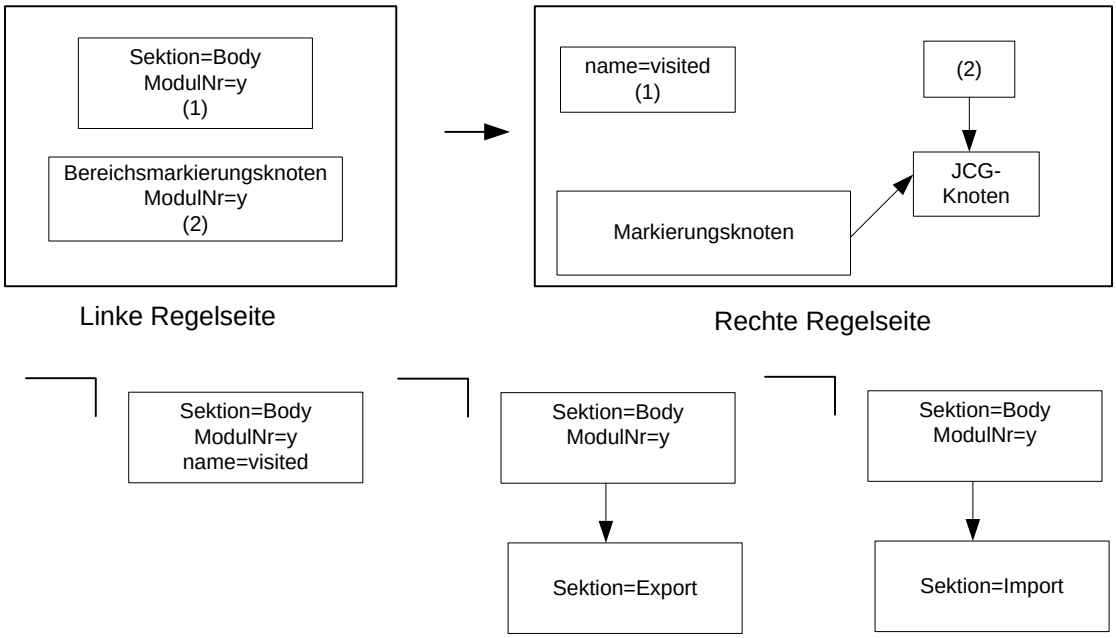


Abbildung 120: Einfügen der Struktur zur Markierung der Körpersektion 1

Die oben dargestellte Regel zeigt die Markierung eines Knotens der Körpersektion unter der Bedingung, dass kein Schnittstellenknoten mit dem zu markierenden Knoten verbunden ist.

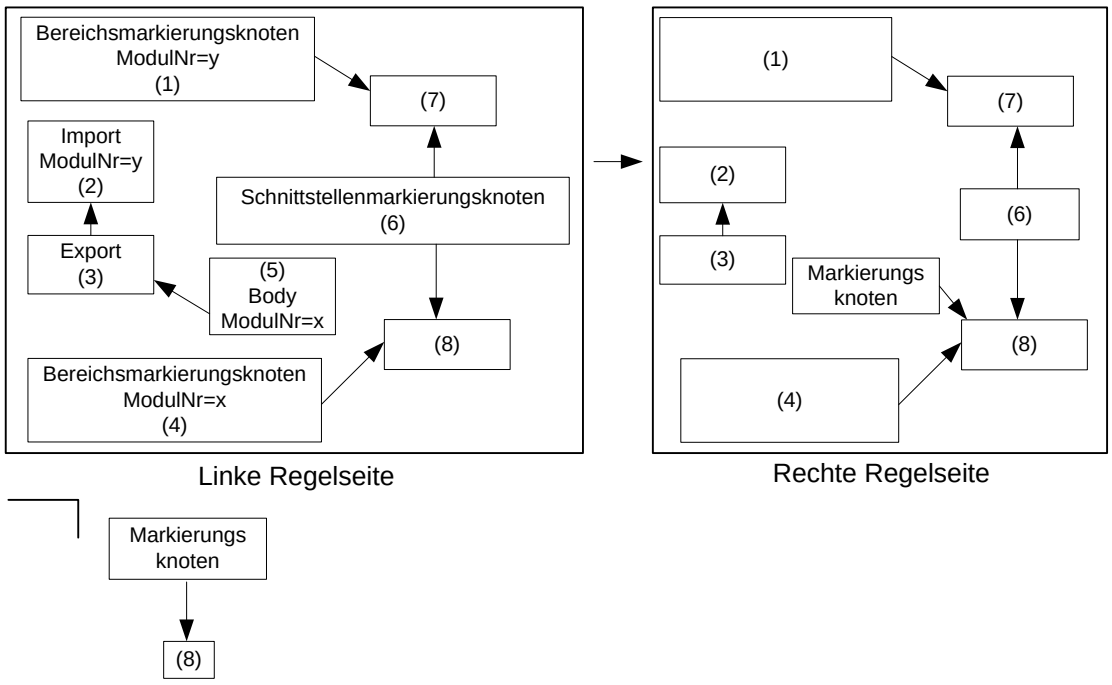


Abbildung 121: Einfügen der Struktur zur Markierung der Körpersektion 2

Die oben dargestellte Regel zeigt die Markierung eines Knotens der Körpersektion unter der Bedingung, dass ein Schnittstellenknoten mit dem zu markierenden Knoten verbunden ist.

Allgemeiner initialer Ableitungsschritt zur Verbindung von JPL-Schnittstellenknoten mit Knoten zur Bereichsmarkierung

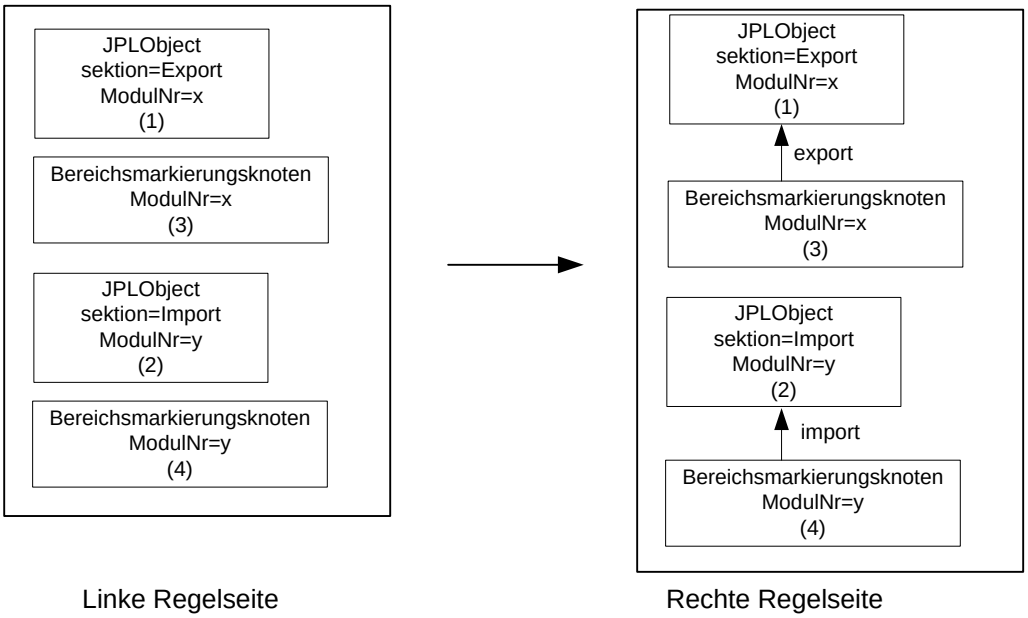


Abbildung 122: Initialer Ableitungsschritt bei JPL-Schnittstellenstruktur

Erstellung von JCG Strukturen über den AJSDG

Kante zwischen *return* und *methodCall*

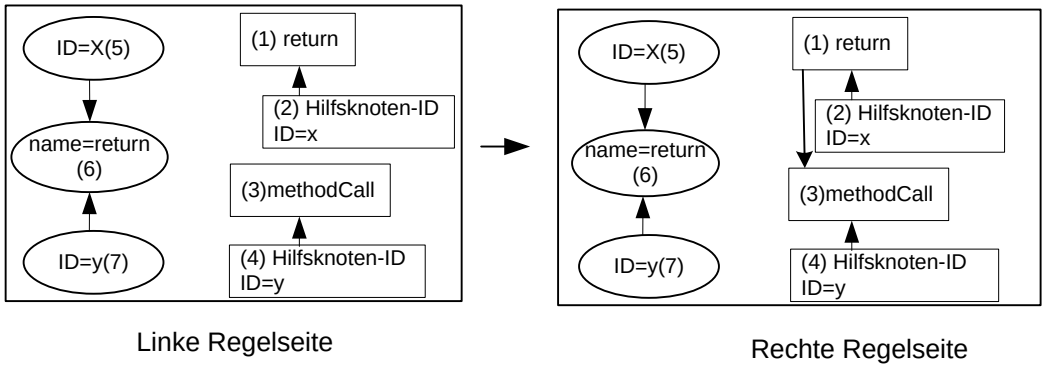


Abbildung 123: Regel zur Erstellung des JCG: Kante zur Variablenrückgabe

Vererbungskante

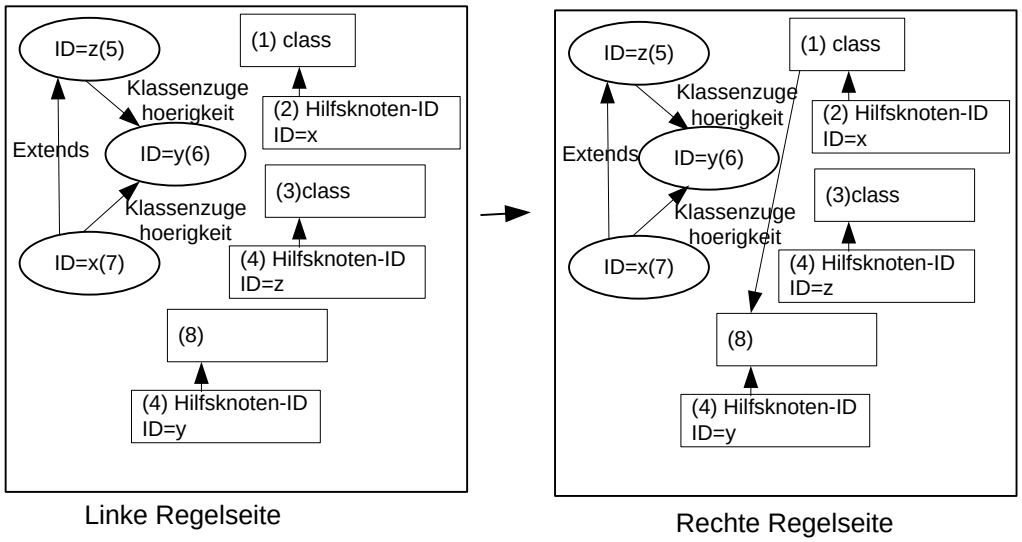


Abbildung 124: Regel zur Erstellung des JCG: Vererbungskante

Technische Hilfsregeln zur Verbindung AJSDG – JCG

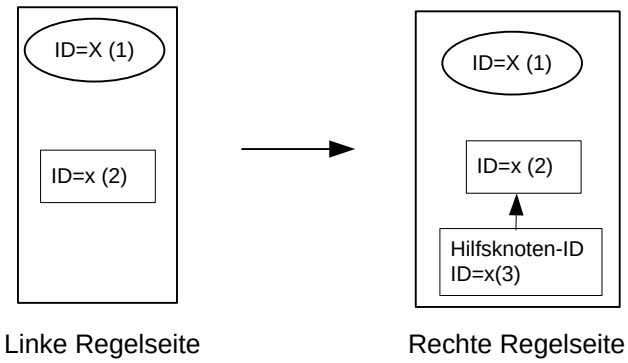


Abbildung 125: Hilfsknoten an JCG-Knoten

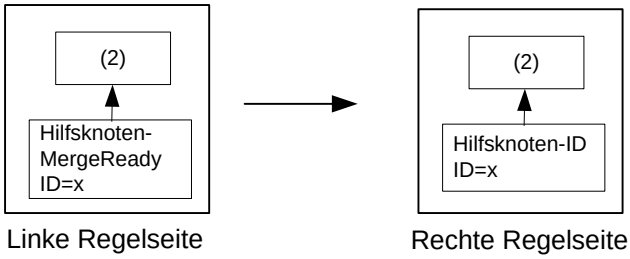


Abbildung 126: Normalisierung der Hilfsknoten 1

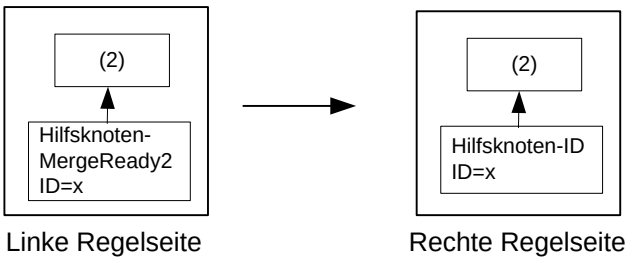


Abbildung 127: Normalisierung der Hilfsknoten 2

Die Normalisierung der Hilfsknoten ist notwendig, da für die Erstellung des AJSDG über den JCG-Quellcodegraphen verschiedene Hilfsknotentypen verwendet wurden, welche für die Mustersuche vereinheitlicht über den Typ *Hilfsknoten-ID* angesprochen werden.

11.3. Regelzuordnung von Anwendungsbeispielen zur Dissertation

Zur Realisierung der in Kapitel 8. beschriebenen Anwendungsbeispiele wurden verschiedene Regelsätze im AGG Tool implementiert. In diesem Kapitel werden die Beziehungen zwischen den implementierten Regeln zu den Regeln, welche in der Arbeit beschrieben werden, tabellarisch dargestellt. Hierbei wird sowohl die Ableitung des abstrakten JPL-Graphen, als auch der Ablauf der Mustersuche betrachtet. Es wird die Beschreibung der einzelnen Regeln in Kombination mit deren Position im Regelablauf dargestellt. Zur praktischen Demonstration der Regelabläufe wurden Videos erstellt welche auf der beiligenden DVD enthalten sind. Weiterhin werden auf der DVD die verwendeten Graphen, Regelsätze, die zur Ausführung notwendigen Tools und Dateien mit Erläuterungen zu den einzelnen Beispielen bereitgestellt.

Für den Ablauf der Regeln zur Ableitung des JPL-Graphen ist folgende generelle Regelablauffolge zu beachten:

1. Technische Regeln zur Ergänzung des Quellgraphen falls notwendig (technische Schritte: Erstellung der Hilfsstrukturen die zur Zuordnung von JCG- und AJSDG-Knoten benötigt werden)
2. Erstellung der Knoten zur Bereichsmakierung und deren Verbindung mit den Knoten der zugeordneten Module
3. Auflösung einer JPL-Schnittstellenknoten-Verbindung (Initiale Schritte der Kapitel 5.5 und 5.6) (Ableitungsschritte)
4. Erstellung der konkreten Suchmustervariante (Ableitungsschritte)
5. Falls weitere JPL-Schnittstellenknoten Verbindungen bestehen gehe zu 3
6. Ableitung der direkten JCG und AJSDG Beziehungen zwischen den Modulen (Ableitungsschritte)

Für den Ablauf der Regeln zur Durchführung der Mustersuche ist folgender genereller Ablauf zu beachten:

1. Normalisierung der Hilfsknoten und Erstellung zusätzlicher JCG-Beziehungen (technische Regeln zur Erstellung des Quellgraphen)
2. Markierung Knoten, die den Modulbereichen zugeordnet werden können (Identifikatorsektion des Moduls)
3. Markierung der Knoten, die diesen Bereichen nicht zugeordnet werden können falls notwendig.
4. Erstellung der transitiven Hülle falls notwendig
5. Suche nach dem Muster

Die Reihenfolge der Regeln wird im Folgenden so dargestellt, wie sie auch in der AGG-Ansicht der Regel erscheint.

Regelzuordnung zum Beispiel: Aufgabe1 Muster1 Parameter eine Bedingung

Regeln zur Ableitung des JPL-Graphen

AGG-Regel	**Regelablauf**	**Regelbeschreibung**
AJSDGImport	S. 74, Prozessschritt 3	212, Abbildung 103
Bereichmarkierung	S. 74, Prozessschritt 1	S. 210, Abbildung 97
JPLPrimitiveAufloesung	Ableitungsschritte für 5.6, angepasst an einzelnen Knoten im Import (sh. rechts) 1	S. 92, Abbildung 39 , angepasst auf einen Knoten im Import
IndirekteUebergabeDurchReturn(Import/Export)	Ableitungsschritte für 5.6, Übergabe durch return, S 94, Schritt 2	S. 212, Abbildung 105
IndirekteUebergabeDurchReturn(Import/Export)1	Ableitungsschritte für 5.6, Übergabe durch return, S 94, Schritt 3	S. 213, Abbildung 106
IndirekteUebergabeDurchReturn(Import/Export)2	Ableitungsschritte für 5.6, Übergabe durch return, S 94, Schritt 4	S. 213, Abbildung 107
IndirekteUebergabeDurchReturn(Import/Export)3	Ableitungsschritte für 5.6, Übergabe durch return, S 94, Schritt 5	S. 95, Abbildung 42 vereinfacht für nur einen JPL-Knoten
AJSDGundJCGidentisch	Technischer Schritt zur Erstellung des Quellgraphen, Prozessschritt 0	S. 221, Abbildung 125
ASJDGBodyanMarkierung	S. 74, Prozessschritt 2	S. 210, Abbildung 98
JCGBodyanMarkierung	S. 74, Prozessschritt 2	S. 210, Abbildung 98
PLPrimitiveImportanMarkierung	S. 74, Prozessschritt 2	S. 211, Abbildung 101

Tabelle 1: Regeln zur Ableitung des JPL-Graphen für „Aufgabe 1 Muster 1 Parameter eine Bedingung“

Regeln zur Ausführung der Mustersuche

AGG-Regel	**Regelablauf**	**Regelbeschreibung**
BereichMethodeMarkieren	S. 107 Prozesschritt 1	S. 206, Abbildung 87
BereichJCGMarkieren	S. 107 Prozesschritt 3	S. 206, Abbildung 88
BereichAJSDGMarkieren	S. 107 Prozesschritt 3	S. 208, Abbildung 92
DatenTransitiveHulle	S. 107 Prozesschritt 5	S. 104, Abbildung 49
KontrollTransitiveHulle	S. 107 Prozesschritt 5	Äquivalent zu S. 104, Abbildung 49
DatenTransitiveHulle2	S. 107 Prozesschritt 5	S, 104, Abbildung 50
KontTransitiveHulle2	S. 107 Prozesschritt 5	Äquivalent zu S, 104, Abbildung 50
Pfadquellenmarkieren	S. 107 Prozesschritt 6	S. 209, Abbildung 95
PfadquellenmarkierenAJSDG	S. 107 Prozesschritt 6	S. 113, Abbildung 54 Ohne Einbezug von Anweisungen auf gleicher

		Hierarhieebene mit der Anweisung des Quellknotens
PfadzielemarkierenAJSDG	S. 107 Prozesschritt 6	S. 113, Abbildung 55
Pfadzielemarkieren	S. 107 Prozesschritt 6	S. 210, Abbildung 96
Pfademarkieren	S. 107 Prozesschritt 6	S. 114, Abbildung 57
Pfademarkieren2	S. 107 Prozesschritt 6	S. 115, Abbildung 58
JCGKnotenausserhalbMarkieren	S. 107 Prozesschritt 3	S. 209, Abbildung 93
AJSDGKnotenausserhalbMarkieren	S. 107 Prozesschritt 3	S. 209, Abbildung 93
SucheImplementierungsvariante	S. 107 Prozesschritt 10	Suchmustervariante, die über die Ableitung des JPL-Graphen erstellt wurde.
KantezwischenreturnundMethodcall	Technische Regel zur Erstellung des JCG, Schritt 0	S. 220, Abbildung 123
HilfsknotenNormalisieren	Technische Regel zur Erstellung des JCG, Schritt 0	S. 221, Abbildung 126
HilfsknotenNormalisieren2	Technische Regel zur Erstellung des JCG, Schritt 0	S. 222, Abbildung 127

Tabelle 2: Regeln zur Ausführung der Mustersuche für „Aufgabe 1 Muster 1 Parameter eine Bedingung“

Regelzuordnung zum Beispiel: Aufgabe1 Muster2GlobalVar mit MarkAll

Regeln zur Ableitung des JPL-Graphen

AGG-Regel	**Regelablauf**	**Regelbeschreibung**
MarkiereAJSDG	S. 74, Prozessschritt 3	S. 211, Abbildung 102
AJSDGundJCGidentisch	Technischer Schritt zur Erstellung des Quellgraphen, Prozessschritt 0	S. 221, Abbildung 125
Bereichsmarkierungsknoteninit	S. 74, Prozessschritt 1	S. 210, Abbildung 97
AJSDGBodyanMarkierung	S. 74, Prozessschritt 2	S. 210, Abbildung 98
JPLMarkierungsknoten_import	S. 74, Prozessschritt 2	S. 211, Abbildung 100
JPLMarkierungsknoten_export	S. 74, Prozessschritt 2	S. 211, Abbildung 99
ASTBodMarkierungsknoten	S. 74, Prozessschritt 2	S. 210, Abbildung 98
IndirekteUebergabeVerschmelzung1	Ableitungsschritte für 5.5 (sh. rechts) 1	S. 82, Abbildung 32, Regel1
IndirekteUebergabeVerschmelzung2	Ableitungsschritte für 5.5 (sh. rechts) 2	S. 82, Abbildung 32, Regel2
IndirekteUebergabeVerschmelzung3	Ableitungsschritte für 5.5 (sh. rechts) 3	S. 82, Abbildung 32, Regel3

IndirekteUebergabeVerschmelzung4	Ableitungsschritte für 5.5 (sh. rechts) 4	S. 82, Abbildung 32, Regel4
IndirekteUebergabeVerschmelzung5	Ableitungsschritte für 5.6 (sh. rechts) 5	S. 98, Abbildung 46, gekürzt, da im importierenden Modul bereits eine Referenz besteht

Tabelle 3: Regeln zur Ableitung des JPL-Graphen für „Aufgabe 1 Muster 2 GlobalVar“

Regeln zur Ausführung der Mustersuche

AGG-Regel	**Regelablauf**	**Regelbeschreibung**
BereichMarkierenSumme	S. 107 Prozesschritt 1	S. 206, Abbildung 87
BereichMarkierenFunktion	S. 107 Prozesschritt 1	S. 206, Abbildung 87
BereichJCGMarkieren	S. 107 Prozesschritt 3	S. 206, Abbildung 88
BereichAJSDGMarkieren	S. 107 Prozesschritt 3	S. 208, Abbildung 92
DatenTransitiveHulle	S. 107 Prozesschritt 5	S. 104, Abbildung 49
KontrollTransitiveHulle	S. 107 Prozesschritt 5	Äquivalent zu S. 104, Abbildung 49
KontrollTransitiveHulle1	S. 107 Prozesschritt 5	Äquivalent zu S. 104, Abbildung 49
DatenTransitiveHulle2	S. 107 Prozesschritt 5	S, 104, Abbildung 50
KontTransitiveHulle2	S. 107 Prozesschritt 5	Äquivalent zu S, 104, Abbildung 50
KontTransitiveHulle3	S. 107 Prozesschritt 5	Äquivalent zu S, 104, Abbildung 50
JCGKnotenausserhalbMarkieren	S. 107 Prozesschritt 3	S. 209, Abbildung 94
AJSDGKnotenMarkieren	S. 107 Prozesschritt 3	S. 209, Abbildung 94
KantezwischenreturnundMethodcall	Technische Regel zur Erstellung des JCG, Schritt 0	S. 220, Abbildung 123
SucheVarianteMuster2Global	S. 107 Prozesschritt 10	Suchmustervariante, die über die Ableitung des JPL-Graphen erstellt wurde
HilfsknotenNormalisieren	Technische Regel zur Erstellung des JCG, Schritt 0	S. 221, Abbildung 126
HilfsknotenNormalisieren2	Technische Regel zur Erstellung des JCG, Schritt 0	S. 222, Abbildung 127

Tabelle 4: Regeln zur Ausführung der Mustersuche für „Aufgabe 1 Muster 2 GlobalVar“

Regelzuordnung zum Beispiel: Aufgabe1 Muster2GlobalVar mit NAC

Regeln zur Ausführung der Mustersuche

AGG-Regel	**Regelablauf**	**Regelbeschreibung**
BereichMethodeMarkierenSumme	S. 107 Prozesschritt 1	S. 206, Abbildung 87, ohne MarkAll Knoten da NACs verwendet werden
BereichMethodeMarkierenFunktion	S. 107 Prozesschritt 1	S. 206, Abbildung 87, ohne MarkAll Knoten da NACs verwendet werden
BereichJCGMarkieren	S. 107 Prozesschritt 3	S. 206, Abbildung 88
BereichAJSDGMarkieren	S. 107 Prozesschritt 3	S. 208, Abbildung 92
DatenTransitiveHulle	S. 107 Prozesschritt 5	S. 104, Abbildung 49
KontrollTransitiveHulle	S. 107 Prozesschritt 5	Äquivalent zu S. 104, Abbildung 49
KontrollTransitiveHulle1	S. 107 Prozesschritt 5	Äquivalent zu S. 104, Abbildung 49
DatenTransitiveHulle2	S. 107 Prozesschritt 5	S, 104, Abbildung 50
KontTransitiveHulle2	S. 107 Prozesschritt 5	Äquivalent zu S, 104, Abbildung 50
KontTransitiveHulle3	S. 107 Prozesschritt 5	Äquivalent zu S, 104, Abbildung 50
SucheVarianteMuster2Global	S. 107 Prozesschritt 10	Suchmustervariante, die über die Ableitung des JPL-Graphen erstellt wurde. Hier werden im Gegensatz zum vorherigen Beispiel zwei NACs verwendet, die unter Nutzung des Templates von S.212, Abbildung 104 erstellt wurden.
HilfsknotenNormalisieren	Technische Regel zur Erstellung des JCG, Schritt 0	S. 221, Abbildung 126
HilfsknotenNormalisieren2	Technische Regel zur Erstellung des JCG, Schritt 0	S. 222, Abbildung 127

Tabelle 5: Regeln zur Ausführung der Mustersuche für „Aufgabe 1 Muster 2 GlobalVar mit NAC“

Regelzuordnung zum Beispiel: Aufgabe1 Muster2 Parameterübergabe zwei Bedingungen

Regeln zur Ableitung des JPL-Graphen

Hinweis zur Spalte **Regelsatz**: In dieser Spalte wird dargestellt durch welchen Ablaufschritt des in Kapitel 7.2. auf Seite 141 dargestellten Prozesses der JPL-Graph abgeleitet wird und welches Template jeweils angesprochen wird. Alle Regeln werden für die Anwendungsbeispiele in einer Datei aufgeführt, so dass der Kopiervorgang zwischen Schritt 1 und Schritt 2 ausgespart wird.

AGG-Regel	**Regelablauf**	**Regelbeschreibung**	**Regelsatz**
AJSDGImport	S. 74, Prozessschritt 3	S. 212, Abbildung 103	Template A Schritt 1
Bereichsmarkierungsknoten	S. 74, Prozessschritt 1	S. 210, Abbildung 97	Template A Schritt 1
JPLPrimitiveAufloesung	Ableitungsschritte für 5.6 (sh. rechts) 1	S. 92, Abbildung 39	Template B Schritt 2
IndirekteUebergabeDurchReturn(Import/Export)1	Ableitungsschritte für 5.6, Übergabe durch return, S 94, Schritt 2	S. 212, Abbildung 105	Template B Schritt 2
IndirekteUebergabeDurchReturn(Import/Export)2	Ableitungsschritte für 5.6, Übergabe durch return, S 94, Schritt 3	S. 213, Abbildung 106	Template B Schritt 2
IndirekteUebergabeDurchReturn(Import/Export)3	Ableitungsschritte für 5.6, Übergabe durch return, S 94, Schritt 4	S. 212, Abbildung 105	Template B Schritt 2
IndirekteUebergabeDurchReturn(Import/Export)4	Ableitungsschritte für 5.6, Übergabe durch return, S 94, Schritt 5	S. 213, Abbildung 107	Template B Schritt 2
IndirekteUebergabeDurchReturn(Import/Export)5	Ableitungsschritte für 5.6, Übergabe durch return, S 94, Schritt 6	S. 95, Abbildung 42	Template B Schritt 2
AJSDGundJCGidentisch	Technischer Schritt zur Erstellung des Quellgraphen, Prozessschritt 0	S. 221, Abbildung 125	Template A Schritt 1
AJSDGBodyanMarkierung	S. 74, Prozessschritt 2	S. 210, Abbildung 98	Template A Schritt 1
JPLPrimitiveExportanMarkierungsknoten	S. 74, Prozessschritt 2	S. 210, Abbildung 98	Template A Schritt 1
JCGBodyanMarkierungsknoten	S. 74, Prozessschritt 2	S. 210, Abbildung 98	Template A Schritt 1
JPLImportanMarkierungsknoten	S. 74, Prozessschritt 2	S. 211, Abbildung 101	Template A Schritt 1

Tabelle 6: Regeln zur Ableitung des JPL-Graphen für „Aufgabe 1 Muster 2 Parameterübergabe zwei Bedingungen“

Regeln zur Ausführung der Mustersuche

Hinweis zur Spalte **Regel einfügen**: In dieser Spalte wird dargestellt durch welchen Ablaufschritt des in Kapitel 7.2. auf Seite 141 dargestellten Prozesses die in der Zeile beschriebene Regel in die Datei zur Mustersuche eingefügt wird und welches Template jeweils angesprochen wird. Die Abfolge der Regeln wird entsprechend der Reihenfolge dargestellt, wie sie nach dem Prozess eingefügt würden. Dies bedeutet die Regeln sind entsprechend ihrer Aufrufreihenfolge angeordnet.

AGG-Regel	**Regelablauf**	**Regelbeschreibung**	**Regel einfügen**
KantezwischenreturnundMethodcall	Technische Regel zur Erstellung des JCG, Schritt 0	S. 220, Abbildung 123	Schritt 7
HilfsknotenNormalisieren	Technische Regel zur Erstellung des JCG, Schritt 0	S. 221, Abbildung 126	Schritt 7
HilfsknotenNormalisieren2	Technische Regel zur Erstellung des JCG, Schritt 0	S. 222, Abbildung 127	Schritt 7
BereichMarkierenSumme	S. 107 Prozesschritt 1	S. 206, Abbildung 87	Schritt 13
BereichMarkierenFunktion	S. 107 Prozesschritt 1	S. 206, Abbildung 87	Schritt 13
BereichJCGMarkieren	S. 107 Prozesschritt 3	S. 206, Abbildung 88	Schritt 15
BereichAJSDGMarkieren	S. 107 Prozesschritt 3	S. 208, Abbildung 92	Schritt 15
JCGKnotenausserhalbMarkieren	S. 107 Prozesschritt 3	S. 209, Abbildung 94	Schritt 22
AJSDGKnotenausserhalb Markieren	S. 107 Prozesschritt 3	S. 209, Abbildung 94	Schritt 22
DatenTransitiveHulle	S. 107 Prozesschritt 5	S. 104, Abbildung 49	Schritt 24
KontrollTransitiveHulle	S. 107 Prozesschritt 5	Äquivalent zu S. 104, Abbildung 49	Schritt 24
KontrollTransitiveHulle1	S. 107 Prozesschritt 5	Äquivalent zu S. 104, Abbildung 49	Schritt 24
DatenTransitiveHulle2	S. 107 Prozesschritt 5	S, 104, Abbildung 50	Schritt 24
KontTransitiveHulle2	S. 107 Prozesschritt 5	Äquivalent zu S, 104, Abbildung 50	Schritt 24
KontTransitiveHulle3	S. 107 Prozesschritt 5	Äquivalent zu S, 104, Abbildung 50	Schritt 24
SucheVarianteMusterMitReturn	S. 107 Prozesschritt 10	Suchmustervariante, die über die Ableitung des JPL-Graphen erstellt wurde.	Schritt 28

Tabelle 7: Regeln zur Ausführung der Mustersuche für „Aufgabe 1 Muster 2 Parameterübergabe zwei Bedingungen“

Regelzuordnung zum Beispiel: Aufgabe 2 Muster 3 LebensmittelGlobalVar

Regeln zur Ableitung des JPL-Graphen

AGG-Regel	**Regelablauf**	**Regelbeschreibung**
Bereichmarkierung	S. 74, Prozessschritt 1	S. 210, Abbildung 97, ohne MarkAll
JPLObjectAufloesung0	Ableitungsschritte für 5.5 (sh. rechts) 1	S. 82, Abbildung 32, Regel 2
ExportMarkierung	S. 74, Prozessschritt 2	S. 210, Abbildung 98
BodyMarkierung	S. 74, Prozessschritt 2	S. 210, Abbildung 98
JPLObjectAufloesung	Ableitungsschritte für 5.5 (sh. rechts) 2	S. 82, Abbildung 32, Regel 1
JPLObjectAufloesung2	Ableitungsschritte für 5.5 (sh. rechts) 3	S. 82, Abbildung 32, Regel 3
JPLObjectAufloesung3	Ableitungsschritte für 5.5 (sh. rechts) 4	S. 82, Abbildung 32, Regel 4
objectDeclarationAufloesung0	Ableitungsschritte für 5.5 - Exkurs (sh. rechts) 1, S. 91	S. 214, initialer Schritt - nicht abgebildet
objectDeclarationAufloesung1	Ableitungsschritte für 5.5 - Exkurs (sh. rechts) 2, S. 91	S. 214, initialer Schritt – nicht abgebildet
objectDeclarationAufloesung2	Ableitungsschritte für 5.5 - Exkurs (sh. rechts) 3, S. 91	S. 214, Abbildung 110
objectDeclarationAufloesung3	Ableitungsschritte für 5.5 - Exkurs (sh. rechts) 4, S. 91	S. 214, Abbildung 111

Tabelle 8: Regeln zur Ableitung des JPL-Graphen für „Aufgabe 2 Muster 3 Lebensmittel GlobalVar“

Regeln zur Ausführung der Mustersuche

AGG-Regel	**Regelablauf**	**Regelbeschreibung**
BereichMethodeMarkierenModulLebensmittellager	S. 107 Prozesschritt 1, bzw. S. 117 Schritt 1	S. 207, Abbildung 89: Bereichsmarkierung für Klassen 1
BereichMethodeMarkierenModulLebensmittel	S. 107 Prozesschritt 1, bzw. S. 117 Schritt 1	S. 207, Abbildung 89: Bereichsmarkierung für Klassen 1
BereichMethodeMarkierenModulLebensmittellagersuchen	S. 107 Prozesschritt 1, bzw. S. 117 Schritt 1	S. 208, Abbildung 91
BereichMethodeMarkierenModulLebensmittellagerif	S. 107 Prozesschritt 1, bzw. S. 117 Schritt 1	S. 208, Abbildung 91
BereichMethodeMarkierenModulLebensmittellagerentnehmen	S. 107 Prozesschritt 1, bzw. S. 117 Schritt 1	S. 208, Abbildung 91

BereichJCGMarkieren	S. 107 Prozesschritt 3, bzw. S. 117 Schritt 1	S. 206, Abbildung 88
VerbindungModul4und3(KlasseLebensmittel--suche)	S. 117 Schritt 2	Aus dem JPL-Graphen abgeleitete Schnittstellenstruktur
VerbindungModul0und1(KlasseLebensmittelliste--entnehme)	S. 117 Schritt 2	Aus dem JPL-Graphen abgeleitete Schnittstellenstruktur
TypVerbindung0und1(KlasseLebensmittel-Instanziierung)	S. 117 Schritt 2	Aus dem JPL-Graphen abgeleitete Schnittstellenstruktur
Modul2Body	S. 117 Schritt 1	Aus dem JPL-Graphen abgeleitete Körperstruktur
KomplettFinish	S. 117 Schritt 3	Aus dem JPL-Graphen abgeleitete Gesamtmusterstruktur

Tabelle 9: Regeln zur Ausführung der Mustersuche für „Aufgabe 2 Muster 3 Lebensmittel GlobalVar“

Regelzuordnung zum Beispiel: Aufgabe3 Muster4 Vererbung

Regeln zur Ableitung des JPL-Graphen

AGG-Regel	**Regelablauf**	**Regelbeschreibung**
AJSDGundJCGidentisch	Technischer Schritt zur Erstellung des Quellgraphen, Prozessschritt 0	S. 221, Abbildung 125,
Bereichmarkierung	S. 74, Prozessschritt 1	S. 210, Abbildung 97, ohne MarkAll
AJSDGBodyanMarkierung	S. 74, Prozessschritt 2	S. 210, Abbildung 98
JPLanMarkierungsknoten_import	S. 74, Prozessschritt 2	S. 211, Abbildung 101
JPLanMarkierungsknoten_export	S. 74, Prozessschritt 2	S. 211, Abbildung 101
JCGBodyanMarkierungsknoten	S. 74, Prozessschritt 2	S. 210, Abbildung 98
IndirekteUebergabeVerschmelzung1	Ableitungsschritte für 5.5 (sh. rechts) 1	S. 82, Abbildung 32, Regel1
IndirekteUebergabeVerschmelzung2	Ableitungsschritte für 5.5 (sh. rechts) 2	S. 82, Abbildung 32, Regel2
IndirekteUebergabeVerschmelzung3	Ableitungsschritte für 5.5 (sh. rechts) 3	S. 82, Abbildung 32, Regel3
IndirekteUebergabeVerschmelzung4	Ableitungsschritte für 5.5 (sh. rechts) 4	S. 82, Abbildung 32, Regel4
IndirekteUebergabeVerschmelzung6	Ableitungsschritte für 5.6 (sh. rechts) 6	S. 98, Abbildung 46 gekürzt, da im importierenden Modul bereits eine Referenz besteht (suche nach Variablenübergabe durch übergeordnete Variable)

IndirekteUebergabeVerschmelzung5	Ableitungsschritte für 5.6 (sh. rechts) 5	S. 84, Abbildung 33, Variante A (suche nach einer Vererbungsstruktur)

Tabelle 10: Regeln zur Ableitung des JPL-Graphen für „Aufgabe 3 Muster 4 Vererbung“

Regeln zur Ausführung der Mustersuche

AGG-Regel	**Regelablauf**	**Regelbeschreibung**
BereichMarkierenModulWuerfel	S. 107 Prozesschritt 1, bzw. S. 117 Schritt 1	S. 207, Abbildung 89: Bereichsmarkierung für Klasse 1
BereichMarkierenModulKonstruktor	S. 107 Prozesschritt 1, bzw. S. 117 Schritt 1	S. 206, Abbildung 87
BereichMarkierenModulKantenlaenge	S. 107 Prozesschritt 1, bzw. S. 117 Schritt 1	S. 208, Abbildung 91
BereichJCGMarkieren	S. 107 Prozesschritt 3, bzw. S. 117 Schritt 1	S. 206, Abbildung 88
BereichAJSDGMarkieren	S. 107 Prozesschritt 3, bzw. S. 117 Schritt 1	S. 208, Abbildung 92
vererbungsbeziehungEtablieren	Technischer Schritt 0	S. 221, Abbildung 124
kantenlaengeberechnenBody	S. 117 Schritt 1	Aus dem JPL-Graphen abgeleitete Körperstruktur
VerbindungKantenLaengeBerechnenKonstruktor	S. 117 Schritt 2	Aus dem JPL-Graphen abgeleitete Schnittstellenstruktur
KomplettFinish	S. 117 Schritt 3	Aus dem JPL-Graphen abgeleitete Gesamtmusterstruktur
HilfsknotenNormalisieren1	Technischer Schritt 0	S. 221, Abbildung 126
HilfsknotenNormalisieren2	Technischer Schritt 0	S. 222, Abbildung 127
DatenTransitiveHulle	S. 107 Prozesschritt 5, bzw. S. 117 Schritt 1	Äquivalent zu S. 104, Abbildung 49
DatenTransitiveHulle2	S. 107 Prozesschritt 5, bzw. S. 117 Schritt 1	S, 104, Abbildung 50

Tabelle 11: Regeln zur Ausführung der Mustersuche für „Aufgabe 3 Muster 4 Vererbung“

Regelzuordnung zum Beispiel: Aufgabe4 Muster5 Lokale Variable

Hinweis zur Spalte **Regelsatz**: In dieser Spalte wird dargestellt durch welchen Ablaufschritt des in Kapitel 7.2. auf Seite 141 dargestellten Prozesses der JPL-Graph abgeleitet wird und welches Template jeweils angesprochen wird. Alle Regeln werden für die Anwendungsbeispiele in einer Datei aufgeführt, so dass der Kopiervorgang zwischen Schritt 1 und Schritt 2 ausgespart wird.

Es werden 3 Templates für Mustervarianten angewendet.

- Template für die direkt spezifizierte direkte Übergabe (B1)
- Template für eine über eine JPL-Struktur spezifizierte direkte Übergabe (B2)
- Template für eine indirekte Übergabe als *return*-Parameter (B3)

Regeln zur Ableitung des JPL-Graphen

AGG-Regel	**Regelablauf**	**Regelbeschreibung**	**Regelsatz**
Bereichmarkierungsknoten	S. 74, Prozessschritt 1	S. 210, Abbildung 97, ohne MarkAll	Schritt 1 Template A
JPLPrimitiveAufloesung1	Ableitungsschritte für einzelen Knoten im Import (sh. rechts) 1	S. 92, Abbildung 39 , angepasst auf einen Knoten im Import	Schritt 2 Template B2 Variante direkt
JPLPrimitiveAufloesung2	Ableitungsschritte für 5.5 (sh. rechts) 2	S. 86, Abbildung 35. Ohne Quell-, Ziel Analyse (nur Korrektur des Gültigkeitsbereichs der Deklaration bei bestehendem Zugriff)	Schritt 2 Template B2 Variante direkt
JPLObjectAufloesungLokaeMembervariable3	Ableitungsschritte für 5.5 (sh. rechts) 3	S. 84, Abbildung 33, Variante B	Schritt 2 Template B2 Variante direkt
JCGExpMarkierungsknoten	S. 74, Prozessschritt 2	S. 210, Abbildung 98	Schritt 1 Template A
JCGBodMarkierungsknoten	S. 74, Prozessschritt 2	S. 210, Abbildung 98	Schritt 1 Template A
JPLImportMarkierungsknoten	S. 74, Prozessschritt 2	S. 211, Abbildung 101	Schritt 1 Template A
objectDeclarationAufloesung1	Ableitungsschritte für 5.5 (sh. rechts) 1, S. 83	S. 214, initialer Schritt - nicht abgebildet	Schritt 2 Template B1 direkte Übergabe
objectDeclarationAufloesung2	Ableitungsschritte für 5.5 (sh. rechts) 2, S. 83	S. 214, initialer Schritt – nicht abgebildet	Schritt 2 Template B1 direkte Übergabe
objectDeclarationAufloesung3	Ableitungsschritte für 5.5 (sh. rechts) 3, S. 83	S. 214, Abbildung 110	Schritt 2 Template B1 direkte Übergabe
objectDeclarationAufloesung4	Ableitungsschritte für 5.5 (sh. rechts) 4, S. 83	S. 214, Abbildung 111	Schritt 2 Template B1 direkte Übergabe
JPLPrimitiveAufloesung	Ableitungsschritte für 5.6 (sh. rechts) 0	S. 92, Abbildung 39	Schritt 2 Template B3 Variante return

IndirekteUebergabeDurchReturn(Import/Export)1	Ableitungsschritte für 5.6, Übergabe durch return, S 94, Schritt 1	S. 212, Abbildung 105	Schritt 2 Template B3 Variante return
IndirekteUebergabeDurchReturn(Import/Export)2	Ableitungsschritte für 5.6, Übergabe durch return, S 94, Schritt 2	S. 213, Abbildung 106	Schritt 2 Template B3 Variante return
IndirekteUebergabeDurchReturn(Import/Export)3	Ableitungsschritte für 5.6, Übergabe durch return, S 94, Schritt 3	S. 212, Abbildung 105	Schritt 2 Template B3 Variante return
IndirekteUebergabeDurchReturn(Import/Export)6	Ableitungsschritte für 5.6, Übergabe durch return, S 94, Schritt 6	S. 214, Abbildung 108	Schritt 2 Template B3 Variante return
IndirekteUebergabeDurchReturn(Import/Export)7	Ableitungsschritte für 5.6, Übergabe durch return, S 94, Schritt 7	S. 214, Abbildung 109	Schritt 2 Template B3 Variante return
IndirekteUebergabeDurchReturn(Import/Export)4	Ableitungsschritte für 5.6, Übergabe durch return, S 94, Schritt 4	S. 213, Abbildung 107	Schritt 2 Template B3 Variante return
IndirekteUebergabeDurchReturn(Import/Export)5	Ableitungsschritte für 5.6, Übergabe durch return, S 94, Schritt 5	S. 95, Abbildung 42 (ohne Umwandung in primitiveDeclaration, da nachfolgend ein Hilfsknoten berücksichtigt werden muss)	Schritt 2 Template B3 Variante return
AJSDGundJCGidentisch	Technischer Schritt zur Erstellung des Quellgraphen, Prozessschritt 0	S. 221, Abbildung 125,	Schritt 1 Template A
AJSDGBodyanMarkierung	S. 74, Prozessschritt 2	S. 210, Abbildung 98	Schritt 1 Template A

Tabelle 12: Regeln zur Ableitung des JPL-Graphen für „Aufgabe 4 Muster 5 Lokale Variable“

Regeln zur Markierung der Graphen in der Körpersektion der einzelnen Module und der Übergabestrukturen für die sektionsbasierte Mustersuche; Datei: *Markierungen in Mustervarianten*

AGG-Regel	**Regelablauf und Regelbeschreibung**
BeichmarkierungVerknuepfen	Kapitel 5.7., Schritt 1
KoerpermarkierungsknotenInit	Kapitel 5.7., Schritt 2
KoerpermarkierungAJSDG	Kapitel 5.7., Schritt 2
KoerpermarkierungJCG	Kapitel 5.7., Schritt 2
IndirekteUebergabeDurchReturn-Markierung	Kapitel 5.7., Schritt 3
WuerfelMemberInberechneFlaeche	Kapitel 5.7., Schritt 3
ContanerMemberInWhile	Kapitel 5.7., Schritt 3
VerbindeReferenzimBodymitUbergabestruktur	Kapitel 5.7., Schritt 3

Tabelle 13: Regeln zur Markierung von Teilgraphen des JPL-Graphen für „Aufgabe 4 Muster 5 Lokale Variable“

Regeln zur Erstellung der Teilgraphen, welche nachfolgend in die Datei zur Mustersuche kopiert werden; Datei: *Markierungen in Mustervarianten*

AGG-Regel	**Regelablauf und Regelbeschreibung**
BehalteKoerpermusterHilfsknoten	Kapitel 5.7., Schritt 5
BehalteKoerpermuster_JCG	Kapitel 5.7., Schritt 5
BehalteKoerpermuster_AJSDG	Kapitel 5.7., Schritt 5
BehalteSchnittstellenmuster	Kapitel 5.7., Schritt 6
BehalteSchnittstellenstruktur	Kapitel 5.7., Schritt 6
BehalteSchnittstellenstrukturAJSDG	Kapitel 5.7., Schritt 6
BehalteSchnittstellenstrukturHilfsknoten	Kapitel 5.7., Schritt 6
LoeseBereichsknotenIndirekteUebergabedurchreturn	Kapitel 5.7., Schritt 6
LoeseBereichsknotenContainerMemberInWhile	Kapitel 5.7., Schritt 6
LoeseBereichsknotenWuerfelMemberInberechneFleache	Kapitel 5.7., Schritt 6

Tabelle 14: Regeln zur Vorbereitung des Kopiervorgangs von Teilgraphen des JPL-Graphen für „Aufgabe 4 Muster 5 Lokale Variable“ in die Datei zum Ablauf der Mustersuche.

Gesamtmuster erstellen für die sektionsbasierte Mustersuche

AGG Regel	**Regelablauf und Regelbeschreibung**
Init	Kapitel 5.7., Schritt 7, Punkt 1
Schnittstellen	Kapitel 5.7., Schritt 7, Punkt 2
SchnittstelleneinKnoten	Kapitel 5.7., Schritt 7, Punkt 2
Body	Kapitel 5.7., Schritt 7, Punkt 3
BodymitExportverknüpfung	Kapitel 5.7., Schritt 7, Punkt 3
BodyMitImportverknüpfung	Kapitel 5.7., Schritt 7, Punkt 3
BodyMitImportverknüpfungImprtAllein	Kapitel 5.7., Schritt 7, Punkt 3
MergeMarkierungen	Kapitel 5.7., Schritt 7, Punkt 4
MergeMarkierungen_SchnittstelleImport	Kapitel 5.7., Schritt 7, Punkt 4

Tabelle 15: Template zur Erstellung des Gesamtmusters für die sektionsbasierte Suche

Regeln zur Ausführung der Mustersuche

Hinweis zur Spalte **Regel einfügen**: In dieser Spalte wird dargestellt durch welchen Ablaufschritt des in Kapitel 7.2 auf Seite 141 dargestellten Prozesses die in der Zeile beschriebene Regel in die Datei zur Mustersuche eingefügt wird und welches Template jeweils angesprochen wird. Die Abfolge der Regeln wird entsprechend der Reihenfolge dargestellt, wie sie nach dem Prozess eingefügt würden. Dies bedeutet die Regeln sind entsprechend ihrer Aufrufreihenfolge angeordnet.

AGG Regel	**Regelablauf**	**Regelbeschreibung**	**Regel einfügen**
HilfsknotenNormalisieren1	Technische Regel 0	S. 221, Abbildung 126	Schritt 7
HilfsknotenNormalisieren2	Technische Regel 1	S. 222, Abbildung 127	Schritt 7
KlassenzugehoerigkeitNormalisiert	Technische Regel 2	Nicht dargestellt, Normalisierung der Klassenzugehörigkeit als Kontrollfluss	Schritt 7
kantezwischenReturnundMethodCall	Technische Regel 3	S. 220., Abbildung 123	Schritt 7
BereichKlasseMarkierenModulWuerfel	S. 107 Prozesschritt 1, bzw. S. 117 Schritt 1	S. 207, Abbildung 89: Bereichsmarkierung für Klassen 1	Schritt 9
BereichKlasseMarkierenModulContainer	S. 107 Prozesschritt 1, bzw. S. 117 Schritt 1	S. 207, Abbildung 89: Bereichsmarkierung für Klassen 1	Schritt 9
ModulGesamtflaeche	S. 107 Prozesschritt 1, bzw. S. 117 Schritt 1	S. 206, Abbildung 87	Schritt 13
BereichAJSDGMarkieren	S. 107 Prozesschritt 3, bzw. S. 117 Schritt 1	S. 208, Abbildung 92	Schritt 15
BereichJCGMarkieren	S. 107 Prozesschritt 3, bzw. S. 117 Schritt 1	S. 206, Abbildung 88	Schritt 15
BereichKantenLaengeMitHierarchiemarkieren	S. 107 Prozesschritt 1, bzw. S. 117 Schritt 1	S. 208, Abbildung 91 (vereinfacht: kein MarkAll-Knoten, da nicht notwendig, sh. JPL-Graph Ableitungsregeln)	Schritt 17
BereichAJSDGMarkierenin Hierarchie	S. 107 Prozesschritt 3, bzw. S. 117 Schritt 1	S. 208, Abbildung 92	Schritt 20
BereichJCGMarkiereninHierarchie	S. 107 Prozesschritt 3, bzw. S. 117 Schritt 1	S. 206, Abbildung 88	Schritt 20
DatenTransitiveHulle	S. 107 Prozesschritt 5, bzw. S. 117	S. 104, Abbildung 49 (nur dieser Schritt auf Grund	Schritt 24

	Schritt 1	von manuellen Graphanpassungen)	
BodyModulberechneFlaeche	S. 117 Schritt 1	Aus dem JPL-Graphen abgeleitete Körperstruktur	Schritt 28, Kapitel 5.7, Schritt 5
BodyModulWhileMitMethodenaufruf	S. 117 Schritt 1	Aus dem JPL-Graphen abgeleitete Körperstruktur	Schritt 28, Kapitel 5.7, Schritt 5
ContainerlMemberInWhile	S. 117 Schritt 2	Aus dem JPL-Graphen abgeleitete Schnittstellenstruktur	Schritt 28, Kapitel 5.7, Schritt 6
WuerfellMemberInberechenFlaeche	S. 117 Schritt 2	Aus dem JPL-Graphen abgeleitete Schnittstellenstruktur	Schritt 28, Kapitel 5.7, Schritt 6
KantenlaengeberechenenIngetKantenlaenge	S. 117 Schritt 2	Aus dem JPL-Graphen abgeleitete Schnittstellenstruktur	Schritt 28, Kapitel 5.7, Schritt 6
Finish	S. 117 Schritt 3	Aus dem JPL-Graphen abgeleitete Gesamtmusterstruktur	Schritt 28, Kapitel 5.7, Schritt 7

Tabelle 16: Regeln zur Ausführung der Mustersuche für „Aufgabe 4 Muster 5 Lokale Variable“

Automatisierungsvideo

Im Automatisierungsvideo wird die halbautomatisierte Mustersuche für das Beispiel:

- Aufgabe1 Muster2 Parameterübergabe zwei Bedingungen

der Dissertation dargestellt. Im Folgenden wird beschrieben welche Schritte automatisch durchlaufen werden, und welche Regeln/Graphen manuell erstellt wurden. Zu beachten ist, dass mit der Implementierung im Rahmen der dargestellten Beispiele nicht das Ziel einer Vollautomatisierung verfolgt wurde, sondern der praktische Einsatz der in der Arbeit dargestellten Regelsätze demonstriert wird. In der Dissertation ist in Kapitel 7.2 das Design eines Tools beschrieben, welches eine vollautomatische Realisierung sowohl der Ableitung des JPL-Musters, als auch des Ablaufs der Mustersuche realisiert.

Der folgende Ablauf orientiert sich am in Kapitel 5, Abbildung 27 dargestellten Prozess.

Erstellung und Ableitung des JPL-Musters:

1. Generierung des JPL-Graphen. Im Beispiel wird zur Erstellung des JPL-Graphen AGG verwendet. In Kapitel 7.2 der Dissertation wird das Design eines Tools erläutert, welches u.a. die Erstellung des JPL-Musters unterstützt und die Ableitung dieses Musters in einen JPL-Graphen ermöglicht.

2. Erstellung der Datei, welche die Regeln zur Ableitung des JPL-Graphen in die konkrete Graphvariante enthält. Diese Datei wurde für das Beispiel manuell erstellt. Für die automatische Implementierung können diese Regelsätze entweder als Templates vorliegen, oder auch dynamisch zusammengestellt werden (sh. Kapitel 7.2).

3. Ausführung der Musterableitung über das Tool RuleControl (Bestandteil der GGX-Toolbox). Der Zielgraph stellt das nachfolgend zu suchende Muster dar.

4. Erstellung der Datei, welche die Regeln zum Ablauf der Mustersuche enthält. Im Beispiel wurde die Datei manuell erstellt. In der automatischen Mustersuche wird diese Datei dynamisch erstellt, basierend auf dem zu suchenden Muster und den hierzu als Voraussetzung notwendigen Strukturen (sh. Kapitel 7.2). Im Beispiel „Aufgabe 1 Muster2 Parameterübergabe zwei Bedingungen“ wird das vollständige Muster als Suchpattern verwendet.

Prüfen des Quellcodes:

5. Erstellung des JCG für den zu untersuchenden Quellcode über das Tool java2ggx.

6. Erstellung des AJSDG über Anwendung der entsprechenden Regelsätze auf den JCG, nachfolgend manuelle Korrektur des AJSDG. Da der AJSDG im aktuellen

Implementierungsstand manuell korrigiert werden muss, wird im Beispiel der bereits vollständig erstellte Graph geladen.

7. Kopieren des JCG und AJSDG in den Regelsatz, der zum Ablauf der Mustersuche verwendet wird. Löschen der leeren Regel Rule, die in jedem automatischen Kopiervorgang erstellt wird.

8. Ablauf der Mustersuche über die AGG Layer-Steuerung.

9. Parsen des Zielgraphen nach dem Ergebnisknoten. Wird dieser gefunden, so wurde das Muster gefunden.

Die Punkte 5,7,8,9 wurden am Lehrstuhl „Spezifikation von Softwaresystemen“ an der Universität Duisburg-Essen für den JCG bereits vollautomatisch implementiert und wurden für die Prüfung einer Vielzahl realer Übungsaufgaben eingesetzt. Während die Schritte in der praktischen Nutzung vollautomatisch hintereinander erfolgen, wurde für dieses Beispiel der Einsatz der verschiedenen Tools zur Verdeutlichung des Ablaufs im Einzelnen dargestellt.

11.4. Inhalt der beiliegenden DVD

Die DVD enthält die in der Dissertation für Kapitel 8 betrachteten und in Kapitel 11.3. referenzierten Anwendungsbeispiele. Für jedes Anwendungsbeispiel sind folgende Informationen enthalten:

- Der Java-Quelltext, in dem das Muster gesucht wird.
- Die Datei *Ableitung des JPL-Graphen.gxx*, welche den JPL-Graphen des jeweiligen Musters und die Ableitungsregeln für den Graphen enthält.
- Die Datei *Ablauf der Mustersuche.ggx*, welche die Graphrepräsentation des Java- Quelltextes und die Regeln zur Mustersuche enthält.
- Eine ggx-Datei, welche die Graphrepräsentation des Java-Quelltexts enthält, die automatisch generiert wurde.
- Falls manuelle Anpassungen an der zuvor genannten Datei notwendig waren, wurde der hieraus resultierende Graph ebenfalls einzeln erfasst.

Für die Beispiele, welche im Ordner *Anwendungsbeispiele mit Videos* enthalten sind, wurden Videos erstellt, die sowohl die Ableitung des JPL-Graphen als auch die Mustersuche beschreiben.

Der Ordner *Anwendungsbeispiele ohne Videos* enthält die weiteren Beispiele, welche im Rahmen der Evaluation in Kapitel 8 betrachtet wurden.

Der Ordner *Tools zur Ausführung der Beispiele* enthält die zur Ausführung der Beispiele notwendigen Applikationen:

- agg_V165: Tool zur Durchführung der Graphtransformationen. Referenz: http://user.cs.tu-berlin.de/~gragra/agg/
- GGxToolbox: Tool u.a. zum Kopieren von Graphen und der Ausführung von Graphregeln. Das Tool wurde am Lehrstuhl „Spezifikation von Softwaresystemen“ an der Universität Duisburg-Essen entwickelt. Referenz: http://www.s3.uni-duisburg-essen.de/research/ggx-toolbox.html
- Java Check:
    - *java2ggx*: Tool zur Transformation von Java Source Code in einen Graphen.
    - *RuleControl*: Tool zur Festlegung von Regelabfolgen.

  Die Tools wurden am Lehrstuhl „Spezifikation von Softwaresystemen“ an der Universität Duisburg-Essen entwickelt.
    - *agg_V124*: Version des AGG-Tools, welches von *java2ggx* verwendet wird.
- SDG: Graphregeln, die in der Bachelorarbeit von Raphael Weiß und Tim Heuer [WH07] an der Universität Duisburg-Essen zur Erstellung des SDG- Graphen entwickelt wurden.

Die Tools wurden unter der Java Version 1.6.0_20 ausgeführt.

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Verwendung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Essen, den 08.09.2014